# Planar Nef Polyhedra and Generic Higher-dimensional Geometry

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Michael Seel

Saarbrücken
8. August 2001

## Abstract

We present two generic software projects that are part of the software library CGAL. The first part describes the design of a geometry kernel for higher-dimensional Euclidean geometry and the interaction with application programs. We describe the software structure, the interface concepts, and their models that are based on coordinate representation, number types, and memory layout. In the higher-dimensional software kernel the interaction between linear algebra and the geometric objects and primitives is one important facet. In the actual design our users can replace number types, representation types, and the traits classes that inflate kernel functionality into our current application programs: higher-dimensional convex hulls and Delaunay tedrahedralisations.

In the second part we present the realization of planar Nef polyhedra. The concept of Nef polyhedra subsumes all kinds of rectilinear polyhedral subdivisions and is therefore of general applicability within a geometric software library. The software is based on the theory of extended points and segments that allows us to reuse classical algorithmic solutions like plane sweep to realize binary operations of Nef polyhedra.



## Zusammenfassung

Wir präsentieren zwei Softwareprojekte, die Teil der Softwarebibliothek CGAL sind. Der erste Teil beschreibt den Entwurf eines Geometriekerns für höherdimensionale euklidische Geometrie und dessen Interaktion mit Anwendungsprogrammen. Wir beschreiben die Softwarestruktur, die auf der Herausarbeitung von Schnittstellenkonzepten und ihren Modellen hinsichtlich Koordinatenrepräsentation, Zahlentypen und Speicherablage beruht. Dabei spielt im Höherdimensionalen die Interaktion zwischen linearer Algebra und den entsprechenden geometrischen Objekten und primitiven Operationen eine wesentliche Rolle. Unser Entwurf erlaubt das Auswechseln von Zahlentypen, Repräsentations- und Traitsklassen bei der Berechnung von $d$-dimensionalen konvexen Hüllen und Delaunay-Simplexzerlegungen.

Im zweiten Teil stellen wir die Realisierung von planaren Nef-Polyedern vor. Das Konzept der Nef-Polyeder umfasst alle linear-polyedrisch begrenzten Unterteilungen. Wir beschreiben ein Softwaremodul das umfassende Funktionalität zur Verfügung stellt. Als theoretische Grundlage des Entwurfs dient die Theorie erweiterter Punkte und Segmente, die es uns erlaubt, vorhandene Algorithmen wie z.B. Plane-Sweep zur Realisierung binärer Operationen von Nef-Polyedern zu nutzen.

# Danke

Diese Arbeit ist das Ergebnis meiner Tätigkeit in der Arbeitsgruppe von Kurt Mehlhorn am Max-Planck-Institut für Informatik zwischen 1996 und 2001. Gigadank an Kurt Mehlhorn für seine Unterstützung, seine motivierende und menschliche Art und die Möglichkeit an den Projekten LEDA und CGAL aktiv teilzunehmen. Die Erfahrungen waren vielfältig und wertvoll: menschlich, ingenieurtechnisch und wissenschaftlich. Neben meinem Doktorvater habe ich auch Stefan Schirra als meinem Zweitbetreuer zu danken. Er hat mir oft genug den C++ Standard interpretiert und mich bei kritischen Fragen beraten.

Die Arbeit in beiden Projektgruppen war ambitioniert, und manchmal voller Emotionen. Die Arbeit innerhalb der LEDA Gruppe mit Kurt, Stefan und Christian war mit ihren kurzen Wegen und schnellen Entscheidungen produktiv und abwechslungsreich. Es war toll, LEDA als wissenschaftliches Projekt zu betreuen. In CGAL ging es mehr als einmal um die perfekte Software. Die CGAL Entwicklertreffen mit Matthias, Herve, Andreas, Bernd, Geert-Jan, Susan, Shai, Michael, Lutz, Dima, Eli, Sylvain, Sven, Monique und Mariette waren kreativ und spannend, wenn auch manchmal die Diskussionen unser aller Geduld strapaziert haben. Wir haben es — trotz Widrigkeiten — geschafft, CGAL zu einem tollen Paket zu machen.

Die Nerds am MPI waren ein tolles Lunch- und Forschungsteam. Danke Andreas, Anker, Ernst, Fritz, Knut, Mark, Stefan (der Schwabe), Sven, Uli, und all den anderen für die erquickliche Zeit. Nicht zu vergessen sind auch all die Menschen in der RGB, Verwaltung und der Bibliothek. Besonderen Dank an Birgit, Bernd, Jörg, Volker (den Grosswesir) und Wolfram. Danke auch Wolfgang, als Quell mathematischer Weisheit bist du nicht zu schlagen.

Zuguter Letzt einen Megadank an Brinja und meine Familie, für all die Unterstützung und Hilfe in den Situationen, als ich zweifelte und haderte. Es hat sich gelohnt.

# Contents

# Chapter 1

# Introduction

This thesis presents research and software engineering in the field of computational geometry. Computational geometry is the scientific field of computer science that tackles geometric problems and provides efficient algorithms for their solution. Our aim is to provide algorithms and implementations thereof that are sound in theory and prove their efficiency in implementations. When implementing software, the last step of packaging and offering it to potential users in the outside world — whether the computer science community or experts from other sciences — is still a major task. This work tries to document the whole process: the software design, the theoretical basics, and the software as part of a software library project.

This thesis describes two software components and their underlying theory. Its results are mainly located in the domain of software engineering but we also present an extension of classical Euclidean geometry that was established in the research for the realization of Nef polyhedra. Our software modules are strongly anchored in and supported by the underlying theory.

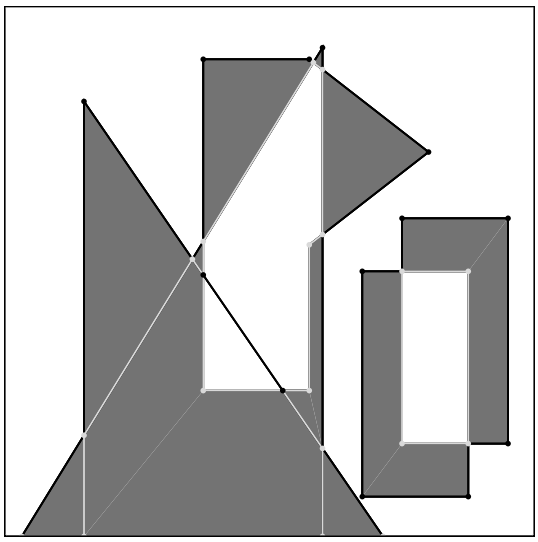


Figure 1.1: A Nef polyhedron.

The practical part of this work had an impact on and is based on the two software libraries LEDA and CGAL. Both are software libraries that offer solutions to problems in the domain of computational geometry. Where LEDA offers an easy but monolithic approach to its data structures and algorithms, more ambitious techniques are used in CGAL. CGAL is designed in the spirit of generic programming and in this sense is also a software engineering experiment that tries to exploit C++ template technology to the extreme. The use of templates allows pattern-based compile-time polymorphism

as opposed to execution-time polymorphism via inheritance and virtual functions. Only template technology offers code composition at compile-time without runtime penalties.

### Higher-dimensional Euclidean Geometry

The first part of the thesis describes a module offering higher-dimensional Euclidean geometry. It describes the objects and primitives that support the development of geometric algorithms in $d$-space. Our contribution is the design of the interface consisting of the objects together with predicates and constructions. Special care was taken to refine the concepts that allow generic adaptation of the kernel from the original monolithic design, *e.g.*, the number types and their docking into the kernel functionality via a linear algebra module. To enhance usability the representation-based kernel families can at the same time be used as traits classes in the instantiation of application classes. The traits classes here encapsulate the geometric primitives that control the logical flow of algorithms. This is one general design pattern of CGAL starting with version 2.3.

### Planar Nef Polyhedra

Our second project concerns the design and implementation of planar Nef polyhedra. The corresponding abstract definition of this polyhedral framework was founded by the Swiss mathematician W. Nef. Our design uses plane maps for the topological description (finite representation) of planar Nef polyhedra. To unify the treatment of the finite and infinite character of the vertices we use extended points as introduced in the third chapter of this thesis. The strength of this design is its clear separation of the special geometric demands of Nef polyhedra from the topological structure used to represent them. We can thereby show that standard affine plane map overlay as presented in the algorithmic part of the second chapter can be used transparently for the solution of the geometrically unbounded but symbolically bounded overlay problem that is part of the binary operations of Nef polyhedra.

### Infimaximal Frames

We introduce the notion of infimaximal frames as an extension of affine geometry. Although it is a vital part in the realization of Nef polyhedra it has also further applications. We present the theory and describe the implementation of three extended kernels as used for the Nef polyhedron software. We use simple but efficient algebraic techniques that aim for different strengths. Two kernels trade efficiency for simplicity. In these kernels, a polynomial ring type is used to realize the arithmetic operations that occur in the implementation of kernel predicates and constructions. Thereby the original geometric complexity with respect to the embedding of affine and frame-supported points is transferred into algebraic complexity that can easily be processed by the polynomial data type. The strength of this implementation is its verifiable correctness. The third kernel trades simplicity for efficiency. We use standard filtering techniques and unroll the occurring algebraic expressions explicitly to obtain a runtime optimized kernel.

### Generic Programming

Generic programming is programming with concepts and models where models are concrete realizations of the abstract concepts. Unfortunately the C++ language constructs and the corresponding compilers only weakly support the checking of whether models fit concepts during the instantiation process.

Rendering generic software modules useful requires a major documentation effort. The reasonable design of the concepts, the correct implementation of their models, but also their expressive documentation are unavoidable requirements for good software design. This thesis tries to describe all facets in a literate programming style. We give the abstract motivation for the design, show the techniques used to implement the modules and cite excerpts from the documentation in the appendix. The full documentation can be consulted on the internet at `www.cgal.org`. The documentation is an important part of our work. It has to guide the users of our software to enable them to compose the right solutions for their problems and especially has to provide recommendations about which models should be used for which purposes. Without the right documentation, the flexibility is a burden and not an asset.

What is gained by genericty? Generic programming allows code extension and adaptation without cut-and-paste programming. Code can easily be tuned and adapted towards a user's needs. Generic composition supports the reuse of software in a well-defined way. Moreover, it allows experimental exchange of models and fast prototyping. We have to admit that it definitely addresses the more sophisticated user, although we provide introductions to CGAL's assets for less advanced users.

```
#include <CGAL/leda_integer.h>
#include <CGAL/Homogeneous_d.h>
#include <CGAL/Convex_hull_d.h>

typedef CGAL::Homogeneous_d<leda_integer> Kernel;
typedef CGAL::Convex_hull_d<Kernel> Convex_hull;

int main()
{
  Convex_hull  H(5);
  Convex_hull::Point_d  p(5);
  while ( std::cin >> p ) H.insert(p);
  Convex_hull::Hull_point_const_iterator it;
  for (it = H.hull_points_begin(); it != H.hull_points_end(); ++it)
    std::cout << *it;
  return 0;
}
```

Figure 1.2: A simple program.

The rest of this chapter is an introduction to the foundations that support the theory and the techniques used in the following chapters. Both projects are based on affine geometry. We therefore first introduce the notions of affine geometry as described by analytical geometry. The results are widely published and we compromise on the proofs to provide just the foundations needed here. The second part of the introduction is about graphs and their embeddings. We use the notions introduced there in the realization of planar Nef polyhedra. There are several UML figures to depict code design in this document. For those who do not know the semantics of UML class diagrams there is a small section introducing them at the end of this chapter.

## 1.1 Spaces

We introduce the concepts of affine geometry as presented in the literature of analytical geometry. The following foundations follow the introduction of the book of W. Nef [Nef78].

We require our readers to have common knowledge of linear algebra. A good treatment can be found in the book of H. Anton [Ant98]. We assume knowledge of the concepts *linear dependence* and *linear independence*, *generating systems*, *bases*, *linear transformations*, and their correspondence to matrices. We start with the relationship between vector spaces and affine spaces.

*vector space* $(F,V)$

A *vector space* $(F,V)$ is a pair of sets where $F$ is a field and $V$ is a non-empty set of *vectors*. For the set $V$ there exists an addition operations such that $(V,+)$ is a commutative additive group. Any element $\lambda \in F$ and any vector $\boldsymbol{v} \in V$ can be combined by a scalar multiplication that returns an element $\lambda\boldsymbol{v}$ in $V$. The scalar multiplication is associative and distributive in the following way: for any $\lambda,\mu \in F$, $\boldsymbol{v},\boldsymbol{w} \in V$ it holds that $\lambda(\mu\boldsymbol{v}) = (\lambda\mu)\boldsymbol{v}$, $\lambda(\boldsymbol{v}+\boldsymbol{w}) = \lambda\boldsymbol{v}+\lambda\boldsymbol{w}$ and $(\lambda+\mu)\boldsymbol{v} = \lambda\boldsymbol{v}+\mu\boldsymbol{v}$ and $1\boldsymbol{v} = \boldsymbol{v}$.

$(\mathbb{R},\mathbb{R}^n)$

In the following we will only consider the canonical vector space $(\mathbb{R},\mathbb{R}^n)$ and we write just $\mathbb{R}^n$ instead of the pair $(\mathbb{R},\mathbb{R}^n)$ above. Its dimension $\mathbf{dim}\,\mathbb{R}^n$ over $\mathbb{R}$ is $n$ and if we omit index ranges for a variable $i$ we will just mean the index range that was specified in the context. Each vector $\mathbf{x}$ is uniquely determined by its coefficients $(x_1,\ldots,x_n)$ with respect to a base $(\mathbf{b}_1,\ldots,\mathbf{b}_n)$ of $V$, i.e. $\mathbf{x} = \sum_i \lambda_i \mathbf{b}_i$.

*Euclidean vector space*

$\mathbb{R}^n$ becomes a (standard-) *Euclidean vector space*, if we define the inner product $(\mathbf{x},\mathbf{y}) := \sum_i x_i y_i$ of two elements $\mathbf{x},\mathbf{y} \in \mathbb{R}^n$ and thereby introduce the concepts of *norm* and *length* of vectors $\|\mathbf{x}\| := \sqrt{(\mathbf{x},\mathbf{x})}$. If $(\mathbf{x},\mathbf{y}) = 0$ then $\mathbf{x}$ and $\mathbf{y}$ are called *orthogonal*.

*affine space*

An *affine space* $(A,V)$ is a pair $(A,V)$ consisting of a non-empty set $A$ of points and a vector space $V$ where the elements of both sets are related via the following properties: (i) each pair $(\mathrm{p},\mathrm{q}) \in A^2$ uniquely determines a vector $\boldsymbol{v} = \overrightarrow{\mathrm{pq}}$. (ii) each point $\mathrm{p} \in A$ and each vector $\boldsymbol{v} \in V$ determine a unique point $\mathrm{q} \in A$ such that $\boldsymbol{v} = \overrightarrow{\mathrm{pq}}$. (iii) for any three points $\mathrm{p},\mathrm{q},\mathrm{r} \in A$ it is required that $\overrightarrow{\mathrm{pq}} + \overrightarrow{\mathrm{qr}} = \overrightarrow{\mathrm{pr}}$. The dimension $\mathbf{dim}A$ of an affine space is the dimension of its underlying vector space $\mathbf{dim}\,V$.

Informally speaking the above definition relates the affine space $A$ to the vector space $V$. This is usually done by fixing a point $\mathrm{o}$ within $A$ called the *origin*. Then each point $\mathrm{p}$ of $A$ corresponds to the translation that moves $\mathrm{o}$ into $\mathrm{p}$. The tranlation is represented by the vector $\overrightarrow{\mathrm{op}}$ of $V$, called the *location vector*. Note that the vector space $\mathbb{R}^n$ can be interpreted as an affine space by defining $\overrightarrow{\mathrm{pq}} := \mathrm{q} - \mathrm{p}$.

*vector* $\mathbf{p}$, *point* $\mathrm{p}$

To separate the linear and affine concepts we write $\mathbf{p}$ for the vector and $\mathrm{p}$ for the affine point object. Whenever affine points meet concepts of linear algebra, we implicitly convert the points into their location vectors.

*linear, affine, and convex combination*

Let $\mathbf{x}_1,\ldots,\mathbf{x}_r \in \mathbb{R}^n$. Any element $\mathbf{x} \in \mathbb{R}^n$ that can be written as

$$\mathbf{x} := \sum_{i=1}^{r} \lambda_i \mathbf{x}_i \quad \lambda_i \in \mathbb{R} \tag{1.1}$$

is called a *linear combination* of $\mathbf{x}_1, \ldots, \mathbf{x}_r$. The linear combination is called *affine* iff $\sum_{i=1}^{r} \lambda_i = 1$. It is called *convex* iff it is affine and additionally $\lambda_i \geq 0$ for all $i = 1, \ldots, r$. Note that the affine and convex linear combinations are concepts of affine spaces. At first glance the linear combination of affine point objects seems irritating. However we can always rewrite an affine linear combination $\mathfrak{x} = \sum_{i=1}^{r} \lambda_i \mathfrak{x}_i$, $\sum_i \lambda_i = 1$ such that the standard intuition comes back: $\mathfrak{x} = \mathfrak{x}_1 + \sum_i \lambda_i (\mathfrak{x}_i - \mathfrak{x}_1)$.

## Linear Subspaces

*linear subspace*

Any non-empty set $L \subseteq \mathbb{R}^n$ that is itself a vector space[1] is called a *linear subspace* of $\mathbb{R}^n$. Its dimension $\dim L$ is defined as the maximal number of linearly independent vectors in $L$. If $L_1, L_2$ are linear subspaces of $\mathbb{R}^n$ then $L_1 + L_2$ and $L_1 \cap L_2$ are also linear subspaces. From linear algebra we know that

$$\dim(L_1 + L_2) + \dim(L_1 \cap L_2) = \dim L_1 + \dim L_2$$

*direct sum* $\oplus$

The sum $L_1 + L_2$ is called the *direct sum* $L_1 \oplus L_2$ iff $L_1 \cap L_2 = \{0\}$.

*linear hull* $\operatorname{lin} A$

For any set $A \subseteq \mathbb{R}^n$, the intersection of all linear subspaces $L \subseteq \mathbb{R}^n$ containing $A$ is the smallest linear subspace that contains $A$.

It is called the *linear hull* of $A$:

$$\operatorname{lin} A := \bigcap_{A \subseteq L} L \tag{1.2}$$

For any non-empty $A \subseteq \mathbb{R}^n$, its linear hull $\operatorname{lin} A$ consists exactly of all linear combinations of the elements of $A$ (if $A = \emptyset$ then $\operatorname{lin} A = \{\mathbf{o}\}$). Especially $\dim \operatorname{lin} A$ is equal to the maximal number of linearly independent vectors in $A$. We always have $0 \leq \dim \operatorname{lin} A \leq n$. The transformation $A \to \operatorname{lin} A$ is a hull operator according to the definition below.

*hull operator*

An operator $\operatorname{op}$ is called a *hull operator* if it fulfills the following conditions for any $S, T \subseteq \mathbb{R}^n$:

H1 $S \subseteq \operatorname{op} S$

H2 $S \subseteq T \Rightarrow \operatorname{op} S \subseteq \operatorname{op} T$

H3 $\operatorname{op} \operatorname{op} S = \operatorname{op} S$

## Affine Subspaces

*flat, affine subspace*

Any pointset $N \subseteq \mathbb{R}^n$ of the form $N = \mathfrak{x} + L$ where $\mathfrak{x} \in \mathbb{R}^n$ and $L$ is a linear subspace of $\mathbb{R}^n$ is called a *flat* (*affine subspace*) of $\mathbb{R}^n$. $L$ is called the linear subspace of the flat $N$, and $\dim N := \dim L$ is its dimension.

*lines, planes, and hyperplanes*

1-, 2-, and $n-1$-dimensional flats are also called *lines*, *planes*, and *hyperplanes*. Affine linear combinations are closed in flats. Moreover for

[1] it is enough to require the set $L$ to be closed under addition and scalar multiplication.

any flat $N$ of dimension $r$ there exists a set $P = \{p_1, \ldots, p_{r+1}\} \subseteq N$ of points such that all affine linear combinations of $P$ span the flat.

*affine (in)dependence*

If $x = \sum_{i=1}^{r+1} \lambda_i p_i, \sum_i \lambda_i = 1$ then $x$ is called *affinely dependent* of $P$ (otherwise *affinely independent*). This definition naturally extends to point sets. A set $A \subseteq \mathbb{R}^n$ is called affinely dependent of $P$ if all elements of $A$ are affinely dependent of $P$.

*barycentric coordinates*

When the $(p_i)_i$ are affinely independent then the $(\lambda_i)_i$ are called the *barycentric coordinates* of $x$ with respect to the points in $P$.

*affine hull* $\mathsf{aff}\, A$

The intersection of flats is empty or again a flat. For any non-empty set $A \subseteq \mathbb{R}^n$ the intersection of all flats $N \subseteq \mathbb{R}^n$ containing $A$ is the smallest flat that contains $A$. It is called the *affine hull* of $A$:

$$\mathsf{aff}\, A := \bigcap_{A \subseteq N} N \tag{1.3}$$

As any linear subspace $L = 0 + L$ is also a flat we deduce that $\mathsf{aff}\, A \subseteq \mathsf{lin}\, A$ (the set of flats containing $A$ is larger than the set of linear subspaces). $\mathsf{aff}\, A$ consists of all affine linear combinations of points in $A$. Conversely, if $\mathsf{dim}\, \mathsf{aff}\, A = r$ then there exist $r+1$ affinely independent points in $A$ that span $\mathsf{aff}\, A$. The transformation $A \to \mathsf{aff}\, A$ is a hull operator. Sometimes the affine dimension of a tuple of points (the dimension of the affine hull of the tuple) is also called their *affine rank*.

## Convex sets

*convex set*

A non-empty set $K \subseteq \mathbb{R}^n$ is called *convex*, if $K$ contains all convex linear combinations of its elements. For any $r$ let $x_1, \ldots, x_r \in \mathbb{R}^n$, $\lambda_1, \ldots, \lambda_r \in \mathbb{R}^{\geq 0}, \sum_i \lambda_i = 1$ we require

$$\sum_{i=1}^{k} \lambda_i x_i \in K \tag{1.4}$$

Convex sets are closed under intersection and linear combination, i.e. for two convex sets $K_1, K_2$ both $K_1 \cap K_2$ and $\lambda_1 K_1 + \lambda_2 K_2$ are again convex. The dimension of a convex set is the dimension of its affine hull $\mathsf{dim}\, K = \mathsf{dim}\, \mathsf{aff}\, K$.

*convex hull* $\mathsf{conv}\, A$

For any set $A \subseteq \mathbb{R}^n$ consider the intersection of all convex sets $K$ that contain $A$. We call this intersection the *convex hull* of $A$:

$$\mathsf{conv}\, A := \bigcap_{A \subseteq K} K \tag{1.5}$$

$\mathsf{conv}\, A$ consists of all convex linear combinations of the elements of $A$. $\mathsf{conv}$ is a hull operator.

## Cones

*positive hull* $\mathsf{pos} A$

The *positive hull* of a set $A \subseteq \mathbb{R}^n$ is defined as $\mathsf{pos} A := \mathbb{R}^+ A := \{\lambda a : \lambda \geq 0, a \in A\}$. $\mathsf{pos}$ is a hull operator.

*cones and apices*

A set $K \subseteq \mathbb{R}^n$ is called *cone with apex* 0, if $K = \mathsf{pos}\, K$. For any $A \subseteq \mathbb{R}^n$

the set $K = \mathbf{pos} A$ is the smallest cone with apex 0 that contains $A$. Now let $K \subseteq \mathbb{R}^n$ and $\mathfrak{x} \in \mathbb{R}^n$. Then $K$ is a *cone with apex* $\mathfrak{x}$, if $K = \mathfrak{x} + \mathbf{pos}(K - \mathfrak{x})$. For any $A \subseteq \mathbb{R}^n$ the set $K = \mathfrak{x} + \mathbf{pos}(A - \mathfrak{x})$ is the smallest cone with apex $\mathfrak{x}$ that contains $A$.

A cone can have many apices. For example any flat $N \subseteq \mathbb{R}^n$ is a cone where all $\mathfrak{x} \in N$ are apices. (For all $\mathfrak{x} \in N$ it holds that $\mathfrak{x} + \mathbb{R}^+(N - \mathfrak{x}) = \mathfrak{x} + \mathbb{R}^+ L = \mathfrak{x} + L = N$.)

If $K$ is a cone with apex $\mathfrak{x}$ and with apex $\mathfrak{y}$ then $K = K + (\mathfrak{y} - \mathfrak{x})$. If $K$ is a cone with apex $\mathfrak{x}$ then the set $T(K)$ of all $\mathfrak{t} \in \mathbb{R}^n$ such that $K + \mathfrak{t} = K$ is a linear subspace of $\mathbb{R}^n$. On the other hand if $L$ is a subspace of $\mathbb{R}^n$ such that $K + L = K$ then $L \subseteq T(K)$. $T(K)$ is the set of translations that map $K$ onto itself. If $K$ is a cone with apex $\mathfrak{x}$ then $\mathfrak{x} + T(K)$ is the set of all apices of $K$.

## Linear functions, hyperplanes and half-spaces

Any *linear function* $f : \mathbb{R}^n \to \mathbb{R}$ is of the form $f(\mathfrak{x}) = (\mathfrak{a}, \mathfrak{x}) + b$, where $\mathfrak{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. If $b = 0$ then $f$ is a *homogeneous* linear function (also called a *linear form*). *linear function f*

If $f$ is not constant, then $F^0 = f^{-1}(0) = \{\mathfrak{x} \in \mathbb{R}^n : f(\mathfrak{x}) = 0\}$[2] is called a *hyperplane* of dimension $n-1$. Any flat $N \subseteq \mathbb{R}^n$ is the intersection of hyperplanes (take all hyperplanes that contain $N$). If $\mathbf{dim} N = m$ then there are always $n - m$ hyperplanes whose intersection is $N$. *hyperplane $F^0$*

If $f : f(\mathfrak{x}) = (\mathfrak{a}, \mathfrak{x})$ is a linear form, then $F^0$ is a linear subspace of $\mathbb{R}^n$ and consists of all vectors $\mathfrak{x}$ that are orthogonal to $\mathfrak{a}$.

We introduce open and closed half-spaces according to the following definitions: *half-spaces $F^+, F^-$*

$$\begin{aligned} F^{\pm} &:= \{\mathfrak{x} \in \mathbb{R}^n : f(\mathfrak{x}) \gtrless 0\} \\ \mathbf{clos} F^{\pm} := F^{\pm} \cup F^0 = \mathbf{cpl} F^{\mp} &= \{\mathfrak{x} \in \mathbb{R}^n : f(\mathfrak{x}) \gtrless 0\} \end{aligned}$$

All of the previously defined half-spaces are convex sets. One can classify points with respect to the above half-spaces. A set $A \subseteq \mathbb{R}^n$ is *(strictly) located on one side* of $F^0$ if

$$f(A)(\gtrless) \gtrless 0 \tag{1.6}$$

We just cite the following general results concerning convex sets and hyperplanes. Let $K \subseteq \mathbb{R}^n$ be convex, $F^0$ a hyperplane, and $K \cap F^0 = \emptyset$ then $K$ is located strictly on one side of $F^0$. If $K_1, K_2 \subseteq \mathbb{R}^n$ are non-empty and convex and $K_1 \cap K_2 = \emptyset$ then there is a non-constant linear function $f$ such that $f(K_1) \geq 0$ and $f(K_2) \leq 0$.

## Affine Transformations

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathfrak{b} \in \mathbb{R}^m$ an $m \times n$ dimensional matrix and an $m$-dimensional vector.

[2] Note that the linear function is a vector space concept but the hyperplane is an affine concept. We implicitly change from points to vectors here.

*affine transformation* $\phi$

Any function of the form $\phi : \mathbb{R}^n \to \mathbb{R}^m$, $\phi(x) = \mathbf{A}x + \mathbf{b}$ is called an *affine transformation*. It is called *homogeneous* if $\mathbf{b} = 0$ and *inhomogeneous* otherwise. It is called *degenerate* if $\mathbf{A}$ does not have full rank ($\det \mathbf{A} = 0$ in case of square matrices), and *non-degenerate* otherwise.

If $\mathbf{A}$ is a regular matrix then $\phi$ is invertible and the inverse $\phi^{-1}$ is again an affine transformation. All non-degenerate affine transformations form a group under composition. A degenerate affine transformation maps the whole space $\mathbb{R}^n$ to a proper subspace of $\mathbb{R}^n$.

Each affine transformation induces a linear transformation $\psi : \mathbb{R}^n \to \mathbb{R}^m$ on the vector space underlying the affine space: $\psi(\mathbf{x}) = \mathbf{A}\mathbf{x}$.

If $\phi$ is non-degenerate then any set of affinely (in)dependent points is mapped to a set of affinely (in)dependent points. Even more interesting, an affine transformation $\phi$ commutes with affine linear combinations, i.e., $\phi(\sum_i \lambda_i x) = \sum_i \lambda_i \phi(x)$.

This implies that a non-degenerate affine transformation preserves barycentric coordinates and maps any $r$-dimensional flat to an $r$-dimensional flat. Moreover, $\phi$ preserves lines, rays, line segments, and parallelism. To describe an affine transformation it is sufficient to fix the images of a maximal affinely independent tuple of points in the domain of the transformation. For any set $A \subseteq N$ it holds that

$$\phi(\operatorname{aff} A) = \operatorname{aff} \phi(A) \tag{1.7}$$

$L_\phi$

Let $\phi : \mathbb{R}^n \to \mathbb{R}^m$ be affine, then we define $L_\phi = \phi^{-1}(\phi(0)) = \{x \in \mathbb{R}^n : \phi(x) = \phi(0)\}$. $L_\phi$ is a linear subspace of $\mathbb{R}^n$. Furtheron, for any $x_1, x_2 \in \mathbb{R}^n$ it holds that $\phi(x_1) = \phi(x_2) \Leftrightarrow x_1 - x_2 \in L_\phi$. Let $K, L \subseteq \mathbb{R}^n, K \oplus L = \mathbb{R}^n$. Then each $x \in \mathbb{R}^n$ can be uniquely represented as a sum $x = k + l$ where $l \in L$ and $k \in K$. Therefore $p : \mathbb{R}^n \to K, p(x) = k$ is a map.

*projection*

We call $p$ a projection from $\mathbb{R}^n$ to $K$ along $L$. $p$ is linear and therefore affine. As $p(x) = x$ for all $x \in K$ it holds that $p^2 = p$. For any subset $A \subseteq \mathbb{R}^n$ it holds that

$$p(A) = (A + L) \cap K \tag{1.8}$$

## Topology

*topology*

Let $X$ be a non-empty set. A system $T$ of subsets of $X$ is called a *topology* on $X$, if $T$ fulfills the following axioms: (1) $X$ and $\emptyset$ belong to $T$. (2) The union of arbitrary many sets of $T$ is in $T$. (3) The intersection of finitely many sets of $T$ is in $T$.
All sets in $T$ are called *open sets*, and the pair $(X, T)$ is called a *topological space*.

*ball* $K_\varepsilon(\mathbf{x})$, *natural topology*

For each $\varepsilon \in \mathbb{R}$ we define the (open) *ball* $K_\varepsilon(\mathbf{x})$ with center $\mathbf{x}$ and radius $\varepsilon$ to be $K_\varepsilon(\mathbf{x}) = \{y \in \mathbb{R}^n : \|y - \mathbf{x}\| < \varepsilon\}$. We then introduce the "*natural topology*" by the introduction of neighborhoods. $U \subseteq \mathbb{R}^n$ is a neighborhood

of $\mathbf{x} \in \mathbb{R}^n$, if there is an $\varepsilon \in \mathbb{R}^+$ such that $K_\varepsilon(\mathbf{x}) \subseteq U$. With this topology $\mathbb{R}^n$ becomes a real topological vector space of dimension *n*. The intersection of two neighborhoods of $\mathbf{x}$ and each superset of a neighborhood of $\mathbf{x}$ is again a neighborhood of $\mathbf{x}$.

*neighborhoods are absorbing*

Each neighborhood $U$ of $\mathbf{0}$ is *absorbing*, i.e., for each $\mathbf{z} \in \mathbb{R}^n$ there exists a $\lambda_0 \in \mathbb{R}^+$, such that $\lambda\mathbf{z} \in U$ for $|\lambda| < \lambda_0$. Accordingly, let $V \subseteq \mathbb{R}^n$ be a neighborhood of $\mathbf{x} \in \mathbb{R}^n$. Then for each $\mathbf{y}_0 \in \mathbb{R}^n$ there is a $\lambda_0 \in \mathbb{R}^+$, such that $\mathbf{y}(\lambda) = \mathbf{x} + \lambda(\mathbf{y}_0 - \mathbf{x}) \in V$ for $|\lambda| < \lambda_0$.

*open, closed set*

A set $A \subseteq \mathbb{R}^n$ is *open* if $A$ is a neighborhood of each $\mathbf{x} \in A$. $A$ is called *closed*, if $\mathsf{cpl}A$ is open.

Any ball $K_\varepsilon$ is open. If $A \subseteq \mathbb{R}^n$ is open (closed) then the same holds for the set $A + \mathbf{t}$, where $\mathbf{t} \in \mathbb{R}^n$. Any union of open sets and any intersection of finitely many open sets is open. Any intersection of closed sets and any union of finitely many closed sets is closed.

*interior* $\mathsf{int}A$

Let $A \subseteq \mathbb{R}^n$. The union of all open subsets of $A$ is called the *interior* (*open kernel*) of $A$:

$$\mathsf{int}A := \bigcup_{S \subseteq A, S \text{ open}} S$$

$A$ is open if and only if $\mathsf{int}A = A$.

*interior point*

Any $\mathbf{x} \in \mathsf{int}A$ is called an *interior point* of $A$. $\mathbf{x} \in A$ is an interior point of $A$ if and only if $A$ is a neighborhood of $\mathbf{x}$.

*closure* $\mathsf{clos}A$

Let $A \subseteq \mathbb{R}^n$. The intersection of all closed supersets of $A$ is called the *closure* (*closed hull*) of $A$:

$$\mathsf{clos}A := \bigcap_{S \supseteq A, S \text{ closed}} S$$

$\mathsf{clos}A$ is the smallest closed superset of $A$. $A$ is closed if and only if $A = \mathsf{clos}A$.

*adherent point*

Any $\mathbf{x} \in \mathsf{clos}A$ is called an *adherent point* of $A$. $\mathbf{x} \in A$ is an adherent point of $A$ if and only if $V \cap A \neq \emptyset$ for any neighborhood $V$ of $\mathbf{x}$.

$\mathsf{clos}$ is a hull operator, whereas $\mathsf{int}$ is not (H2, H3 are valid, but $\mathsf{int}A \subseteq A$ instead of H1). We can deduce the following:

$$\mathsf{int}\,\mathsf{cpl}A = \mathsf{cpl}\,\mathsf{clos}A \quad , \quad \mathsf{clos}\,\mathsf{cpl}A = \mathsf{cpl}\,\mathsf{int}A$$
$$\mathsf{int}\,(A \cap B) = \mathsf{int}A \cap \mathsf{int}B \quad , \quad \mathsf{clos}\,(A \cup B) = \mathsf{clos}A \cup \mathsf{clos}B$$

*exterior* $\mathsf{ext}A$ *and boundary* $\mathsf{bd}A$

The exterior of a point set $A \subseteq \mathbb{R}^n$ is defined as the interior of its complement $\mathsf{ext}A := \mathsf{int}\,\mathsf{cpl}A$. The boundary is defined to be $\mathsf{bd}A := \mathsf{clos}A \cap \mathsf{clos}\,\mathsf{ext}A$. The interior, boundary and closure of a set $A \subseteq \mathbb{R}^n$ are related via $\mathsf{clos}A = \mathsf{int}A \cup \mathsf{bd}A$.

Linear functions are continuous on topological vector spaces. For any continuous map the preimage of an open (closed) set is an open (closed) set. Therefore any half-space $F^\pm = f^{-1}(\mathbb{R}^\pm)$ is an open set. A hyperplane $F^0 = f^{-1}(0)$ is closed. Moreover the complement of a half-space $\mathsf{cpl}F^\pm$ is closed.

*induced topology*

We can transfer the global topology of $\mathbb{R}^n$ to a subset $M \subseteq \mathbb{R}^n$ by the restriction to $V(\mathbf{x}) \cap M$ for any neighborhood $V$ of an element $\mathbf{x} \in \mathbb{R}^n$. Then

$M$ is a topological subspace of the topological space $\mathbb{R}^n$. The topology of $M$ is called *induced* by $\mathbb{R}^n$ on $M$.

*relative neighborhood, relative interior, relatively open*

The topological concepts of subspaces are prefixed by the word *relative*, e.g. $V(\mathbf{x}) \cap M$ is called the relative neighborhood of $\mathbf{x}$ in $M$. Accordingly, a subset $A \subseteq M$ is called relatively open (closed) in $M$ if $A$ is open (closed) with respect to the induced topology. Moreover the relative interior $\mathsf{relint}\, A$ is the interior of $A$ with respect to the induced topology. We often use the relative interior in the special case where $M = \mathsf{aff}\, A$. To say that $A$ is *relatively open* in this sense means with respect to the flat $\mathsf{aff}\, A$.

For any flat $M$ and point $x \in M$ a relative neighborhood $U(x)$ has the property that $\mathsf{aff}\, U = M$. If $K \subseteq \mathbb{R}^n$ is convex, then $\mathsf{clos}\, K$ and $\mathsf{relint}\, K$ are also convex. If $K \subseteq \mathbb{R}^n$ is non-empty and convex then $\mathsf{relint}\, K$ is also non-empty. For a convex set $K \subseteq \mathbb{R}^n$ we can show that $\mathsf{relint}\, K = \mathsf{relint}\, \mathsf{clos}\, K$, $\mathsf{clos}\, K = \mathsf{clos}\, \mathsf{relint}\, K$, and $\mathsf{aff}\, \mathsf{relint}\, K = \mathsf{aff}\, K = \mathsf{aff}\, \mathsf{clos}\, K$.

*bounded set*

A set $A \subseteq \mathbb{R}^n$ is called *bounded*, if there exists a $\rho \in \mathbb{R}^+$ such that $\|x\| < \rho$ for any $x \in A$.

If $A$ is bounded then any $B \subseteq A$, $\mathsf{int}\, A$, $\mathsf{relint}\, A$, $\mathsf{clos}\, A$, $\mathsf{conv}\, A$ are also bounded. The intersection of bounded sets and the union of finitely many bounded sets are bounded. If $A$ and $B$ are bounded then so is $A + B$ (in particular $A + \mathbf{x}$ for any $\mathbf{x} \in \mathbb{R}^n$). Any finite set $A \subseteq \mathbb{R}^n$ is bounded.

## 1.2 Graphs and Plane Maps

Our representation of the concept of graphs and plane maps is based on the notions from the LEDA book [MN99].

*graph G*

A (directed) *graph* $G = (V, E)$ consists of a set of nodes $V$ and a set of edges $E \subset V \times V$. We say an edge $e = (v, w)$ has *source* $v$ and *target* $w$. We often use a more functional notation $v = \mathit{source}(e)$. For each node $v$ there is a list of edges $A(v)$ called its adjacency list that contains all edges with $\mathit{source}(e) = v$.

*map M*

A *map* $M$ is a *bidirected edge-paired* graph. This means for each edge $e = (v, w)$ there exists a reversal edge $e' = (w, v)$ and there exists a bijective map *twin* such that $\mathit{twin}(e) = e'$ and $\mathit{twin}(e') = e$. The pairwise reversal edges $e$, $e'$ are called *twins*. Sometimes we consider edge pairs $(e, e')$ as undirected edges (*uedge*) of a corresponding undirected graph.

*drawing I of a graph*

A *drawing* $I$ of a graph $G$ into a surface $S$ is an assignment of coordinates (on the surface) to the nodes in $V$ and of parametrized curves on $S$ to the edges $e$ in $E$ such that the curve starts at $\mathit{source}(e)$ and ends at $\mathit{target}(e)$.

*embedding of a graph*

A drawing of a graph on a surface $S$ is called an *embedding* if the images of edges contain no images of nodes in their relative interiors, if the images of edges belonging to distinct uedges are disjoint except for endpoints, and if the curves assigned to edges belonging to the same uedge are reversals of each other.

*planar map M*

An embedding into the plane is called a *planar embedding* and a planar

embedding in which every edge is mapped to a straight line segment is called a *straight line embedding*. A map $M$ is called *planar* if it has a planar embedding.

*order-preserving embedding*

For each node $v$ of $G$ the adjacency list $A(v)$ has a cyclic order. An embedding is called *order-preserving* if for every node $v$ the counterclockwise ordering of the curves $I(e), e \in A(v)$, around $I(v)$ agree with the cyclic ordering of the edges in $A(v)$.

The embedding of a node $v$ into the plane is specified by the operation $I(v) = \mathit{point}(v)$. Then the straight line embedding of each edge $e$ is equal to $I(e) = \mathit{segment}(e)$ which is the segment spanned by the points $\mathit{point}(\mathit{source}(e))$ and $\mathit{point}(\mathit{target}(e))$. Embedded nodes are also called *vertices*.

*faces and boundary cycle*

The embedding $I$ of $G$ into the plane partitions the plane into open maximal connected point sets. Each such point set is called a *face*. A face is completely described by the uedges and nodes whose embeddings form the boundary of the face. Since uedges are incident to at most two faces, we simplify the incidence description if we assign one face $f$ to each directed edge $e$ (as part of the uedge). We specify that a directed edge is incident to the face on its left side by $f = \mathit{face}(e)$, and we say that $e$ is part of the *boundary cycle* of $f$. A boundary cycle is a collection of edges such that their embedding forms one connected component of the boundary of $f$. The cyclic order of the edges in the boundary is defined such that $f$ is always on the left side of the edges.

*face cycle*

The order of $A(v)$ is defined by two maps. Let $e = (v,w)$, $e' = (v,w')$. If $e'$ is the successor of $e$ in $A(v)$ then $e' = \mathit{cyclic\_adj\_succ}(e)$ and $e = \mathit{cyclic\_adj\_pred}(e')$. For iteration around the face cycle we define $\mathit{next}(e) = \mathit{cyclic\_adj\_pred}(\mathit{twin}(e))$ and $\mathit{previous}(e) = \mathit{twin}(\mathit{cyclic\_adj\_succ}(e))$. By this symmetric *previous* – *next* relationship all edges are partitioned into the face cycles. Two edges $e$ and $e''$ are in the same face cycle if $e = \mathit{next}^*(e'')$. All edges $e$ in the same face cycle have the same adjacent face $f = \mathit{face}(e)$. One can show that a face cycle is exactly the boundary cycle of the incident face if the embedding is order preserving ([MN99, Lemma 46]). We thereby match the combinatorial concept "face cycle" with the topological concept "boundary cycle".

Note that the relationship between the order on adjacency lists and the movement in face cycles was determined by two geometric fixations. The order-preservation property is based on a counterclockwise ordering of the embedded edges around $v$ and each oriented edge is incident to the face on its left side.

As a consequence, when designing a data structure to represent an order-preserving embedding, one can choose to maintain either the adjacency lists or the correct definition of the face cycles by the *previous-next* relation. The LEDA graph is centered around the former concept. The CGAL halfedge data structure, on the other hand, is centered around the face cycle concept. Both approaches can lead to the same interface, albeit via quite different implementation techniques.

*plane map P*

A *plane map* $P = (V,E,F)$ is a map $M = (V,E)$ with a fixed cyclic order on the adjacency lists, an *order-preserving* embedding, and *enriched by face objects*. The embedded map $M$ is called the 1-skeleton of $P$. Each face in $F$ refers to a maximal connected subset of the plane bounded by the embedding of the edges and nodes of the 1-skeleton. The size of a plane map is the number of nodes, edges and faces.

Although the adjacency lists have a circular structure there is an operation that provides an entry point into $A(v)$. For each vertex $\mathit{first\_out\_edge}(v)$ determines the start for an iteration through $A(v)$. Thereby, the adjacency list can also be interpreted as a sequence starting with a dedicated edge.

In standard planar Euclidean geometry two nodes can be ordered via their embedding by the lexicographic comparison of the coordinate entries. We additionally qualify the structure of the adjacency lists of a node.

*forward-prefix of $A(v)$*

An edge $e$ is called *forward-oriented* if $\mathit{point}(\mathit{source}(e)) <_{lex} \mathit{point}(\mathit{target}(e))$. We say that the adjacency list $A(v)$ has a *forward-prefix* if only the first part of $A(v)$[3] consists of forward-oriented edges (possibly zero). The counterclockwise order-preserving embedding implies that the slopes of the lines that support the edges in the prefix increase with their position in the prefix. Similarly, we define *backward-oriented* edges and the *backward-suffix* of $A(v)$ (the slopes of the supporting lines of the edges of the backward-suffix are also increasing).

*attributed plane maps*

In our context plane maps have attribues, each object $u \in V \cup E \cup F$ carries an information attribute $\mathit{mark}(u)$.

*subdivision and support*

For two objects *a*, *b* that represent subsets of points in the plane we say that *a supports b* if the point set of *b* is contained in the point set of *a*.

Let $S$ be a set of segments in the plane where we also allow trivial segments. We subdivide any segment $s$ into its endpoints and its relative interior. The arrangement of the segments in $S$ partitions the plane into disjoint point sets with respect to a *support relation*. These point sets are: the endpoints of the segments, the points in the non-degenerate intersection of the relative interior of two segments, the points in subsegments of the relative interior of the segments in $S$ (*supported* by at least one segment), and the points in the complement of the union of all segments in $S$.

When we store this structure we are interested not only in the point sets (points, relatively open segments) but also in their topology. Therefore, we choose to represent the arrangement using a plane map $P = (V,E,F)$ as described above. For each endpoint of a segment in $S$ and each point of intersection of two segments we obtain one vertex in $V$ (identical coordinates imply only one vertex). Between any two vertices we have one uedge if there is a segment that *supports* the convex combination of the vertices, and there is no other vertex contained therein. Note that there can be more than one segment supporting such uedges if some segments overlap. In between the 1-skeleton constructed from these objects lie faces which are not supported by any segment from $S$. Note that the uedges represent relatively open one-dimensional point sets, and the faces represent open

[3] $A(v)$ interpreted as a sequence starting in $\mathit{first\_out\_edge}(v)$.

two-dimensional sets.

Now we go one step further. Let $P_i = (V_i, E_i, F_i), i = 0, 1$ be two plane map structures. The overlay of two plane maps $P_0$, $P_1$ is the plane map $P$ representing the subdivision of the plane obtained by interpreting the skeleton objects of the $P_i$ according to their embedding as trivial and non-trivial segments, constructing the overlay of these segments and adding the faces.

## 1.3 Programming Concepts

Our work interacts with the following three programming libraries and their concepts. LEDA (Library of Efficient Data Types and Algorithms) is a C++ library of combinatorial and geometric data types and algorithms [LEDa]. CGAL is the Computational Geometry Algorithms Library that is developed by the ESPRIT projects CGAL and GALIA. The library is written in C++ and follows the idea of generic programming [CGA]. STL is the Standard Template Library. The library is part of the ISO C++ standard [com98].

*generic programming, concepts and models*

The algorithms of the STL together with the concepts of iterator and container form a triad that supports *generic programming*. The paradigm builds on data abstraction, but takes it in a different direction from that of object-oriented programming. In particular generic programming uses *concepts* to provide abstract descriptions of the required interface between algorithms and data types. Any types that realize these requirements serve as *models* of the concept.

The template facility in C++ makes programming in this generic way possible. The template parameters of an algorithm or data structure act as concepts and, as long as the type used in the template has all of the members and methods that the template uses, the template can be instantiated. This frees polymorphism from inheritance. A typical example of a generic algorithm from the STL has an interface that uses iterators.

*iterators*

*Iterators* are a generalization of pointers that allow a programmer to work with different data structures (containers) in a uniform manner. An iterator is the glue that allows one to write a single implementation of an algorithm that will work for data contained in an array, a list or some other container - even a container that did not yet exist when the algorithm was implemented. An iterator is a concept, not a programming language construct. It can be seen as a set of requirements. A type is an iterator if it satisfies those requirements. In this sense, a pointer to an element of an array is an iterator.

We introduce the following notation that allows us to talk about the sets, sequences, and tuples referenced by an iterator range. For an iterator range $[\mathit{first}, \mathit{last})$ we define $S = \mathit{tuple}\ [\mathit{first}, \mathit{last})$ as the ordered tuple $(S[0], S[1], \ldots S[d-1])$ where $S[i] := *{+}{+}^{(i)}\mathit{first}$ (the element obtained by forwarding the iterator by operator $++$ $i$ times and then dereferencing it to get the value to which it points). Accordingly, $\mathit{set}\ [\mathit{first}, \mathit{last})$ is the un-

ordered version of *tuple*. We write *size* $[first, last) := d$. If we index the tuple as above then we require that $++^{(d)} first == last$ (note that *last* points beyond the last element to be accepted).

*circulators*

*Circulators* are quite similar to iterators but are defined to work on circular data structures like a ring list in a uniform manner. Thus the main difference is the absence of a last element. Please note that circulators are not part of the STL, but of CGAL. For the complete description of the requirements please refer to the CGAL Reference Manual, Part 3 [SVY00].

*traits classes*

The original concept of traits classes [Mye95] uses template class specialization as a technique to associate certain types and functions to a type model. There, the type model is the template parameter of the traits class. Specializations of a general traits template class allow one to fix types and functions for the model.

In the domain of software libraries that require more elaborate concepts, the notion of traits classes has evolved. A traits class can bundle disjoint concepts into one larger unit. Thereby it focuses the genericity at one spot which simplifies documentation, maintainance, and usability. In CGAL most generic algorithms and abstract data types can be adapted by means of a traits class. Note that a traits class model can fit different concepts and can thereby be usable in more than one algorithmic instantiation.

*smart pointers*

We often use the smart pointer programming pattern. Smart pointers realize copy by reference via a pointer and a transparent dynamic memory allocation scheme maintained via reference counting. We give an example[4] in Figure 1.3. Assume we want to realize a $d$-dimensional point type storing a tuple of $d$ entries of a number type *NT*.

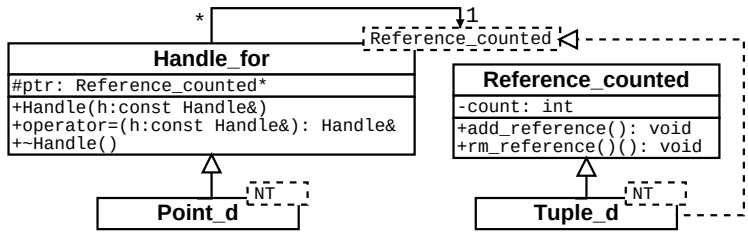



Figure 1.3: Implementing a $d$-dimensional point with a smart pointer.

In CGAL the types supporting smart pointers are called *Handle_for*<*T*> and *Reference_counted*. The former is a base class for the handle type realizing the visible interface and the latter is a base class (a model for the type parameter *T* of *Handle_for*<*T*>) realizing the maintainance operations needed by the handle base class. Realizing a point by a smart pointer pattern is easy now. Just derive the storage container of the tuple *Tuple_d*<*NT*> from *Reference_counted* and derive the interface type *Point_d*<*NT*> from

[4] A short introduction to the notions of the UML follows at the end of this concepts chapter.

*Handle_for*< *Tuple_d*<*NT*> >. Now any object of type *Point_d*<*NT*> is a handle to a *d*-tuple.

We shortly describe the behaviour in case of construction, access, and destruction. A point is constructed by giving its base class a storage candidate. The counter of the reference-counted object is initialized to one.

```
Point_d(const Tuple_d<NT>& t) : Handle_for< Tuple_d<NT> > (t) {}
```

Any copy construction or assignment is covered by code from the base classes. A copy construction basically redirects the smart pointer to the corresponding reference-counted tuple object and increases the reference count (by *add_reference*( )).

```
Point_d(const Point_d<NT>& p) : Handle_for< Tuple_d<NT> > (p) {}
```

Any destruction ∼*Point_d*( ) of a handle object decreases the reference count (by *rm_reference*( )). The reference-counted tuple object is destroyed as soon as the counter reaches zero.

What are the advantages? In case of many assignment or copy construction operations, these operations boil down to pointer redirection and counter incrementation. As soon as the storage class is larger than a certain threshold this pattern pays off. Runtime tests have shown that this threshold is reached at around four words on a SUN machine. Below this size, memory copy mechanisms are faster than the maintainance overhead. Note that access to a handled object always suffers from one redirection. For example the following operation realizes access from a handle object to the *i*th entry of a tuple object.

```
NT operator[](int i) { return ptr->entry(i); }
```

Thus there is a trade-off involved. The low-dimensional kernels of CGAL therefore come in two flavors (handled and non-handled). For larger objects the smart pointer scheme is recommended. Note that for programming languages like JAVA reference based copy construction is the default.

A *decorator* is a programming pattern [GHJV95] that attaches additional functionality to an object (or several objects). The ability to transparently combine different sets of functionality to one object via different decorators is the main advantage compared to simple subclassing. Decorators are used in our implementation of Nef polyhedra. *decorator*

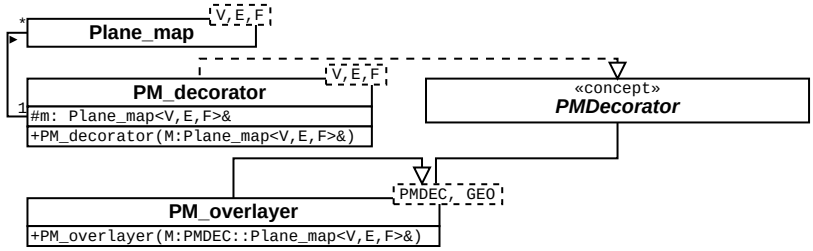



Figure 1.4: Implementing a decorator on top of a plane map data type.

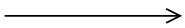
We modularized functionality for manipulating plane maps into several decorator classes. The classes obtain generic type information via traits classes. We provide a simplified picture in Figure 1.4. The plane map data type is parameterized by the types for its vertices, edges, and faces. The top level decorator *PM_decorator*<$V,E,F$> adds functionality and defines an interface to an object of type *Plane_map*<$V,E,F$> referenced via *m*. The reference is initialized on construction, *e.g.*, by the constructor parameter *M*. In some cases the user is not able to access *m* directly to protect the internal representation of the plane map (read-only access). All decorators also provide all handle and iterator types for access to the plane map in their local scope. Decorators extending the functionality further are obtained by subclassing. For example, *PM_overlayer*<$PMDEC, GEO$> extends the simple decorator with functionality to calculate the overlay of two plane maps. The necessary geometric kernel functionality is added by the template parameter *GEO*. By subclassing the interface of the base class is transferred to the subclass.

## 1.4 Unified Modeling Language

*UML*

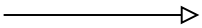
The Unified Modelling Language (UML) is an abstract framework for describing the structure of object-oriented software modules. Though it has at its core a strong semantics that can even be used for code generation, for us it is mainly a vehicle to describe software structure and to develop a road map of the software project. We mainly use class diagrams of the UML to depict the concepts and models that we use to offer a generic software module realizing Nef polyhedra in the plane. The following constructs are part of our figures.

*association*



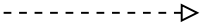
A directed association. The type at the source of the arrow uses the type at the tip of the arrow.

*generalization*



A generalization. The type at the source of the arrow generalizes (is inherited from) the type at the tip of the arrow.

*realization*



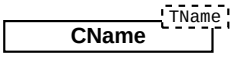
A realization. In the language of concepts and models, the type at the source of the arrow is a model that realizes the concept at the tip of the arrow.

*template typename association*



A template typename-concept association. The template typename at one end of the association must fit the requirements specified by the concept at the other end of the association on instantiation.

*class template*




A class template *template* < *typename TName* > *class CName*.

*concept*


A concept *CName*. The concept defines the type and method requirements of a template parameter.

Class diagrams additionally document attributes and methods. Both are written in a PASCAL-like notation where identifiers precede their types.

Protection mechanisms are marked by "+" for public access, "−" for private access, and "#" for protected access. For further information on the UML please resort to the standard literature, e.g., [RJB99].

All referenced research reports that have been published as part of our research can be obtained from the WWW server of the MPI [mpi].

# Chapter 2

# Generic higher-dimensional geometry

In this chapter we describe the design of the dynamic *d*-dimensionalkernel of CGAL. That kernel realizes higher-dimensional affine geometry where the dimension is an interface parameter of the geometric objects[1]. Our presentation is complete with respect to software design but not with respect to functionality. We will not present a complete implementation description of all objects, primitives, or interface details, but concentrate on the relevant aspects. An earlier and complete implementation description can be found as a technical report [MMN+96]. The project was supported by many people and many design ideas stem from discussion within the CGAL consortium. The application layer is originally due to M. Müller and J. Ziegler [MZ94], the kernel design is based on the original proposal presented at the SCG conference [MMN+97] that was in turn influenced by the LEDA geometric kernel. The final design transformation was developed in coorporation with S. Schirra, H. Brönnimann, S. Pion, and M. Hoffmann. See also the design paper [HHK+01a, HHK+01b].

## 2.1 Motivation

There have already been several overview papers about the motivation for CGAL and its design. N. Amenta [Ame97] gives an overview on the state-of-the-art of computational geometry software before CGAL and provides many references. Computational geometry software was intensively discussed at the First ACM Workshop on Applied Computational Geometry, cf. [Meh96, Lee96, Ove96]. A. Fabri et al. [FGK+96] report about the basic design of the lower dimensional kernel. Precision and robustness issues of a computational geometry library are discussed in [Sch99]. In the US, an implementation effort with a goal similar to that of the CGAL-project has been started at the Center for Geometric Computing, located at Brown University, Duke University, and John Hopkins University. They state their goal as an effective technology tranfer from Computational Geometry to relevant applied fields. Their geometry library is called GeomLib [TV97] and is implemented in Java.

There are several collections of computational geometry software modules called gems. Gems carry the characteristic of being specialized in a small field of computational geometry. They are

[1] as opposed to a possible template approach via number template arguments.

often easy to use but due to their diverse origins non-homogeneous in their handling. The striking fact is that many reimplement geometric functionality to a certain extent that is already available somewhere. But their nature forces the design to be self-contained and thus application design in the gem world mostly starts by creating a specialized geometric kernel for the needs of the application.

The development of a generic geometric software library tries to overcome this shortcoming of the gem approach. Application design should be able to start from a collection of geometric objects and geometric primitives like predicates and constructions.

One application gem in the field of computational geometry that found wide application is the quickhull[2] package of C. Barber, D. Dobkin, and H. Hudhanpaa [BDH96]. That software is available on a number of platforms but is also only designed for a special application domain: the package is robust but overcomes rounding errors by only calculating an approximation of the convex hull of a set of points that guarantees some consistency conditions. There is no object oriented geometric interface, and it is mostly blackbox-usage of functionality. The whole geometric framework used in the blackbox is hidden from the user and therefore the quickhull package is mostly used as one stage of a pipelined software process.

LEDA [MN99] offers monolithic designed 2-dimensional and 3-dimensional geometric kernels. Actually, the d-dimensional kernel started as a LEDA extension package (in the spirit of LEDA) and evolved in parallel to the CGAL project.

The geometric kernels of LEDA are also offered in the Cartesian and homogeneous fashion. However, LEDA is based on precompiled libraries, and as such there is no generic parameterization of number types. The advantages are very straightforward: usage is easy and tuning has been done on top of the monolithic design for the average application domain. The generic approach of CGAL introduces more freedom to the user as well as to the designer of the kernel. This means also more work for both: at least some type definitions to instantiate the corresponding kernel for the user and a clean concept specification plus coping with cutting edge compiler technology for the designer. However, users of CGAL gain the possibility to tune and test the consequences of different combinations of representation and number types for the corresponding application domain. The possibility of experimental optimization is the advantage of generic geometry kernels.

## 2.2 Kernel Design

Higher-dimensional affine geometry is strongly connected to its mathematical treatment (linear algebra and analytical geometry). Therefore a central task is the implementation and integration of a linear algebra module in a generic way meeting the design goals of CGAL. We will shortly present the central ideas of our accurate implementation and will describe some features of the concept interface.

The development of this kernel and in parallel the lower dimensional kernels lead to two instantances of the kernel as an abstract concept. In 2- and 3-space the instantiations realize the two flavors of coordinate representation: *Cartesian* and *Homogeneous*. These representation types are strongly related to the topic of number types. As long as we restrict ourselves to geometric problems that can be solved within the rational domain we avoid the complications of irrational numbers and the need for number types that allow the corresponding calculations. Our kernel is mainly designed to solve problems that are restricted to the rational domain.

Number types that represent fields (or approximations thereof) can be used to store points and vectors in Cartesian representation and algebraic terms are allowed to contain division operations. When coordinate inputs are chosen from a grid and the algebraic terms used are closed within the

[2] we mention this one as representative for the software gems available over the internet.

rational field, one can also resort to a homogeneous representation of integer components. Our kernel development is based on multiprecision integer number types and supports homogeneous coordinates. We just call the kernel including its representation style *Homogeneous_d*. The corresponding Cartesian counterpart *Cartesian_d* is also available and completes the two-fold design as realized in the lower dimensional kernels.

The objects of a geometric kernel naturally relate to the notions presented in our introduction 1.1. We will give an overview of the existing objects, possible extensions and describe the design features that make our code efficient.

Finally we will describe the integration of two applications into CGAL that use the higher-dimensional geometry: convex hulls and nearest and furthest site delaunay triangulations. We will sketch the application idea and mainly report on the interaction between application code and kernel primitives.

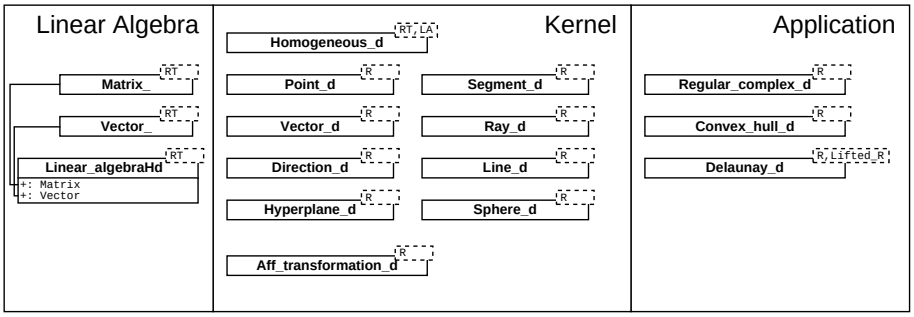



Figure 2.1: The class types involved in the kernel: *Linear_algebraHd*<*RT*> is a model of our linear algebra concept that fits the *LA* parameter of *Homogeneous_d*<*RT*,*LA*>. *Homogeneous_d*<*RT*,*LA*> is a model of the kernel concept *R*.

## Coordinate representation

Cartesian coordinates are based on an orthonormal coordinate system (an orthogonal frame whose axis obtain their scale based on the standard base $(\boldsymbol{e}_0,\ldots,\boldsymbol{e}_{d-1})$ where $\boldsymbol{e}_i = (\delta_{i,j})_j$). In a Cartesian coordinate system the coefficients of a point (or vector) are taken from a field $F$. Ideally, (and in theory) we take $F = \mathbb{R}$, but as soon as we write programs we can only approximate $\mathbb{R}$, *e.g.*, the data type double is a rational (grid based) approximation of real numbers that is also only a bounded approximation of the rational field. The problem of rounding errors is an intrinsic problem when using doubles in the absence of reals in geometric algorithms. Even though there are application areas where using doubles and tolerating failure is acceptable, one of the main objectives of CGAL is robustness (combined with flexibility). As in higher dimensions the accumulated rounding errors are even more a source of failure, the development of the kernel was primarily based on the homogeneous representation of multiprecision rational numbers that lead to accurate geometric implementations.

So let us consider a point $\mathfrak{p} = (p_0,\ldots,p_{d-1}) \in \mathbb{R}^d$. In the standard mathematical treatment the $(p_i)_i$ are called the Cartesian coordinates of $\mathfrak{p}$. They are the coefficients of the geometric vector[3] of $\mathfrak{p}$ with respect to the standard base $(\boldsymbol{e}_0,\ldots,\boldsymbol{e}_{d-1})$. If our point has only coefficients from the field of rational numbers then we can also resort to the homogeneous representation $\mathfrak{p} = (h_0,\ldots,h_{d-1},h_d)$ such

[3] the vector that maps the origin of the coordinate system into $\mathfrak{p}$.

that $p_i = h_i/h_d$. Note that the homogeneous representation in general is not unique (we can multiply the homogeneous representation by a common factor without changing the cartesian coordinates.). The advantage of homogeneous representation in higher dimensions is multiple. Compared to the Cartesian representation using $d$ explicit quotients it saves space. Moreover, many expressions that appear in predicates are simpler due to the common denominator structure and as such their evaluation is also faster. And finally, the homogeneous representation avoids divisions and exact division might either be not an operation of the number type or at least a more expensive operation than multiplication. Number types that can be used for the homogeneous components are multiprecision integer types like *leda_integer* or the *CGAL::Gmpz*[4], but also *leda_reals* (where the division indeed is more expensive than multiplication).

## Concepts and Models

Generic programming is programming up to concepts. The original geometric kernel [MMN+96] was a well-modularized but closely integrated programming module. The port to CGAL lead to the identification of the conceptual interfaces that separate the different modules and made the actual code modules models of the concepts. Several concepts are part of the higher-dimensional geometry: the ring type *RT*, the linear algebra type *LA*, and the kernel (representation type) *R*. The kernel concept can be seen in two views:

**global view** – the global kernel objects like *Point_d<R>* (cf. Figure 2.1) together with the corresponding predicates, constructions, and the kernel objects' method interface make up an intuitive easy-to-use kernel interface that is available to write application programs and experiment with different representation classes or number types.

**traits view** – the representation class *R* itself defines the concept of a general traits class that is used in the application layer of CGAL as a generic interface to kernel functionality. This concept is mainly defined around function objects and has more the flavor of functional programming.

In either case *Homogeneous_d<RT,LA>* fits both concepts as a model. Instantiation of geometric objects like *Point_d< Homogeneous_d<RT,LA> >* allows a user to use the global kernel view. Using *Homogeneous_d<RT,LA>* as a traits class, *e.g.*, as in *Convex_hull_d< Homogeneous_d<RT,LA> >*, makes use of the traits concept.

The main advantage of this design is the transparent usage of representation classes. In the first version of CGAL most applications had their own traits classes and users and maintainers had to handle them separately. In this final design maintainance is much easier, but also application design can be based on the ready-to-use kernel traits models.

## Linear Algebra

The realization of *Homogeneous_d* is strongly connected to the concept of linear algebra. The concept interface can be found in the Appendix 4.1.1. The default model of *LA* is bundled in the class *Linear_algebraHd<RT>*.

The core operation of *Linear_algebraHd<RT>* is a Gaussian elimination scheme for a non-homogeneous linear system $Ax = b$ as described by J. Edmonds [Edm67]. For a recent reference

[4] equals GNU multiprecision integer type wrapped by a C++ class interface

see the books of A. Schrijver [Sch86, part I] or C. Yap [Yap97, lecture X]. We present a literate programming realization of the Gaussian elimination algorithm. Those not interested in the details can skip this presentation and jump to section 2.3.

Consider a linear system $A \cdot x = b$. Gaussian elimination operates in two phases. In the first phase it transforms $A$ by a sequence of row and column operations into an upper diagonal matrix and in the second phase it solves the resulting upper diagonal system.

Let us have a closer look at the first phase. It operates in subphases, numbered 0 to $m-1$ where $m$ is the number of rows of $A$. Before the $k$-th subphase we have transformed $A$ into a matrix

$$C_k = \begin{bmatrix} D & E \\ 0 & F \end{bmatrix},$$

where $D$ is a non-singular upper triangular matrix of order $k$. For $k = 0$ we have $C_0 = A$. In the $k$-th subphase we first determine a nonzero element in $F$ (called the *pivot-element*) and move it into the left upper corner of $F$ by interchanging rows and columns if necessary and then subtract suitable multiples of the top row of $F$ from the other rows of $F$ to zero out all below diagonal entries in $F$'s first column. To simplify the presentation we assume that no interchanging of rows and columns is ever necessary.

**Lemma 2.2.1:** All entries of $F$ can be written as rational numbers with denominator $\det A_k$ where $A_k$ is the submatrix formed by the first $k$ rows and columns of $A$.

*Proof.* For a matrix $M$ and row indices $i_1, \ldots, i_k$ and column indices $j_1, \ldots, j_l$ use $M_{i_1,\ldots,i_l}^{j_1,\ldots,j_l}$ to denote the $l \times l$ submatrix formed by the elements in the selected rows and columns. Consider any entry $f_{ij}$ of $F$. We have

$$f_{ij} = \frac{\det\left((C_k)_{0,\ldots,k-1,i}^{0,\ldots,k-1,j}\right)}{\det\left((C_k)_{0,\ldots,k-1}^{0,\ldots,k-1}\right)}$$

since the matrix in the numerator has $f_{ij}$ in the right lower corner, zeroes in all other entries of the last row, and the matrix in the denominator in its other row and columns. The numerator is therefore $f_{ij}$ times the denominator. Observe next that both determinants do not change if we write $C_0$ instead of $C_k$ since $C_k$ is obtained from $C_0$ by subtracting multiples of the first $k$ rows from the other rows. Thus,

$$f_{ij} = \frac{\det\left((C_0)_{0,\ldots,k-1,i}^{0,\ldots,k-1,j}\right)}{\det\left((C_0)_{0,\ldots,k-1}^{0,\ldots,k-1}\right)}$$

□

The Lemma above suggests to take $\det A_k$ as the denominator of all entries of $F$, to store the denominator separately, say in a variable *denom*, and to keep only the numerators in the matrix $F$. Thus, we maintain the invariants

$$\begin{array}{ll} (1) & denom = \det A_k \\ (2) & F_{ij} = \det A_{0,\ldots,k-1,i}^{0,\ldots,k-1,j}, \text{ for } i \geq k, j \geq k \\ (3) & f_{ij} = F_{ij}/denom. \end{array}$$

The effect of the $k$-th subphase is to replace $f_{ij}$ by

$$\begin{aligned} f'_{ij} &= f_{ij} - f_{ik} \cdot f_{kj} / f_{kk} \\ &= (f_{ij} \cdot f_{kk} - f_{ik} \cdot f_{kj}) / f_{kk} \\ &= ((F_{ij} \cdot F_{kk} - F_{ik} \cdot F_{kj}) / \det A_k) / F_{kk} \end{aligned}$$

for $i > k$ and $j \geq k$.

**Lemma 2.2.2:** $F_{kk} = \det A_{k+1}$ and the division $(F_{ij}F_{kk} - F_{ik}F_{kj})/\det A_k$ is without remainder.

*Proof.* Invariant (2) shows

$$F_{kk} = \det(A^{0,\ldots,k-1,k}_{0,\ldots,k-1,k}) = \det A_k$$

and Lemma 1 tells us that $f'_{ij}$ can be written as a rational number with denominator $F_{kk}$. Thus, $(F_{ij}F_{kk} - F_{ik}F_{kj})/\det A_k$ must be an integer. □

We summarize. We take a matrix $C$ and initialize it with $A$. Before the $k$-th subphase we have

(1) $C_{ii} = \det A_{i+1}$ for $0 \le i \le k$
(2) $C^{0,\ldots,k-1}_{0,\ldots,k-1}$ is non-singular upper triangular
(3) $C_{ij} = 0$ for $i \ge k$ and $j < k$
(4) $C_{ij} = \det A^{0,\ldots,k-1,j}_{0,\ldots,k-1,i}$ for $i \ge k, j \ge k$
(5) $C_{ij} = \det A^{0,\ldots,i-1,j}_{0,\ldots,i-1,i}$ for $i < k, j > i$

In the $k$-th subphase we set

$$C_{ij} = (C_{ij}C_{kk} - C_{ik}C_{kj})/\det A_k$$

For $i > k$ and $j \ge k$. The division is without remainder.

So far we ignored pivoting and the right-hand side $b$. We handle $b$ by adjoining it to $A$ as an additional column. We handle pivoting by storing all column interchanges (there is no need to keep track of the row interchanges since we adjoined $b$ to $A$ and hence handle both sides of the equation in the same way). We store the column interchanges in an array *var*: What is now column $j$ was originally column $var[j]$. In other words, column $j$ of $C$ represents variable $var[j]$.

It is useful to keep track of all row operations performed. We do so in a matrix $L$ which we initialize to an $m \times m$ identity matrix and subject to the same row operations as $C$, i.e., in the $k$-th subphase we set

$$L'_{ij} = (L_{ij} \cdot C_{kk} - C_{ik} \cdot L_{kj})/\det A_k$$

for $i > k$ and all $j$. This maintains the invariant

$$L \cdot (A|b) \cdot P = C$$

where $(A|b)$ denotes the matrix obtained by adjoining $b$ to $A$, $P$ is the permutation matrix corresponding to *var* and $C$ is the matrix we are working in. We initialize $C$ with $(A|b)$ and we intialize $P$ with the identity matrix.

We still need to fill in two details in our treatment of phase 1: Why do the entries of $L$ stay integral and how do we discover unsolvability?

**Lemma 2.2.3:** The entries of $L$ stay integral.

*Proof.* Let us again ignore pivoting. Then

$$L = \begin{bmatrix} U & 0 \\ V & W \end{bmatrix}$$

before subphase $k$ where $U$ is a lower diagonal matrix of order $k$ and $W$ is a diagonal matrix. It is easily proved by induction on $k$ that all diagonal entries of $W$ are equal to $\det A_k$, i.e., $W = (\det A_k) \cdot I$, and that the diagonal entries of $U$ are $\det A_0$, $\det A_1$, ..., $\det A_{k-1}$. Let

$$A = \begin{bmatrix} A_k & \dots \\ A'_k & \dots \end{bmatrix},$$

then $U \cdot A_k = D$ and $V \cdot A_k + W \cdot A'_k = 0$. Thus, $U = DA_k^{-1}$ and $V = -WA'_kA_k^{-1} = -(\det A_k) \cdot A'_kA_k^{-1}$. This implies already the integrality of $V$ since all entries of $A_k^{-1}$ are rational numbers whose denominator divides $\det A_k$ (by Cramers's rule). For matrix $U$, we at least know that it is uniquely defined. It therefore suffices to show that there is an integral matrix $U$ satisfying $U \cdot A_k = D$. This is easy to see: We have $D_{ij} = \det A^{0,\dots,i-1,j}_{0,\dots,i-1,i}$ and expansion according to the last column yields

$$D_{ij} = \sum (-1)^{l+j} \det A^{0,\dots,i-1}_{0,\dots,l-1,l+1,\dots,i} A_{lj}.$$

Thus, $U_{ij} = (-1)^{l+j} \det A^{0,\dots,i-1}_{0,\dots,l-1,l+1,\dots,i}$. □

Unsolvability is easy to detect. We discover unsolvability in subphase $k$ when the search for a nonzero pivot is unsuccessful but the current right hand side has a nonzero entry in row $k$ or below.

We are now ready for the program.

⟨*implementing Linear_algebra*⟩≡
```
template <class RT>
bool Linear_algebraHd<RT>::
linear_solver(const Matrix& A, const Vector& b,
              Vector& x, RT& D, Matrix& spanning_vectors, Vector& c)
{
  bool solvable = true;
  ⟨initialize C and L from A and b⟩
  ⟨phase 1⟩
  ⟨test whether a solution exists and compute c if there is no solution⟩
  if (solvable) {
    ⟨compute solution space⟩
  }
  return solvable;
}
```

We start with the initialization. The matrix in which we will calculate is $C = (A|b)$. $L$ is initialized to the identity matrix.

⟨*initialize C and L from A and b*⟩≡
```
int i,j,k; // indices to step through the matrix
int rows = A.row_dimension();
int cols = A.column_dimension();
CGAL_assertion_msg(b.dimension() == rows,
  "linear_solver: b has wrong dimension");
Matrix C(rows,cols + 1), L(rows); // zero initialized
for(i=0; i<rows; i++) {
  for(j=0; j<cols; j++)
    C(i,j)=A(i,j);
```

```
    C(i,cols)=b[i];
    L(i,i) = 1;
  }
```

In phase 1 of Gaussian elimination we know that column $j$ of $C$ represents the $var[j]$th variable. The array is indexed between 0 and $cols - 1$.

⟨*phase 1*⟩≡
```
  std::vector<int> var(cols);
  for(j=0; j<cols; j++) var[j] = j;
  _RT denom = 1; // the determinant of an empty matrix is 1
  int sign = 1;  // no interchanges yet
  int rank = 0;  // we have not seen any nonzero row yet
  for(k=0; k<rows; k++) {
    ⟨search for a nonzero element; if found it is in (i, j)⟩
    if ( non_zero_found ) {
      rank++; //increase the rank
      ⟨interchange rows k and i and columns k and j⟩
      ⟨one subphase of phase 1⟩
    }
    else
      break;
  }
```

We search for a nonzero element.

⟨*search for a nonzero element; if found it is in* $(i, j)$⟩≡
```
  bool non_zero_found = false;
  for(i = k; i < rows; i++) { // step through rows $k$ to |rows - 1|
    for (j = k ; j < cols && C(i,j) == 0; j++) ;
    // step through columns |k| to |cols - 1|
    if (j < cols) {
      non_zero_found = true;
      break;
    }
  }
```

We interchange rows and columns. Any exchange changes the sign of the determinant.

⟨*interchange rows k and i and columns k and j*⟩≡
```
  if (i != k) {
    sign = -sign;
    /* we interchange rows |k| and |i| of |L| and |C| */
    L.swap_rows(i,k); C.swap_rows(i,k);
  }
  if (j != k) {
    /* We interchange columns |k| and |j|, and
       store the exchange of variables in |var| */
    sign = - sign;
```

```
    C.swap_columns(j,k);
    std::swap(var[k],var[j]);
  }
```

Now we are ready to do the pivot-step with the element $C_{k,k}$. We do the $L$'s first since we want to work with the old values of $C$. Note that the division by *denom* is allowed as this factor is contained in the nominator term.

⟨*one subphase of phase 1*⟩≡
```
  for(i = k + 1; i < rows; i++)
    for (j = 0; j <  rows; j++)  //and all columns of |L|
      L(i,j) = (L(i,j)*C(k,k) - C(i,k)*L(k,j))/denom;
  for(i = k + 1; i < rows; i++) {
    /* the following iteration uses and changes |C(i,k)| */
    RT temp = C(i,k);
    for (j = k; j <= cols; j++)
      C(i,j) = (C(i,j)*C(k,k) - temp*C(k,j))/denom;
  }
  denom = C(k,k);
  ⟨check invariant L·A·P = C⟩
```

It is good custom to check the state of a computation. The matrix $L$ multiplied by $A$ permuted as given by *var* should be equal to the current $C$. The permutation *var* moves column $var[j]$ of $A$ to column $j$ of $C$.

⟨*check invariant L·A·P = C*⟩≡
```
  #ifdef CGAL_LA_SELFTEST
  for(i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++) {
      RT Sum = 0;
      for (int l = 0; l < rows; l++)
        Sum += L(i,l)*A(l, var[j]);
      CGAL_assertion_msg( Sum  == C(i,j),
        "linear_solver: L*A*P different from C");
    }
    RT Sum = 0;
    for (int l = 0; l < rows; l++)
      Sum += L(i,l)*b[l];
      CGAL_assertion_msg( Sum  == C(i,cols),
        "linear_solver: L*A*P different from C");
  }
  #endif
```

We are done with Gaussian elimination. At this point $C$ has a *rank* × *rank* upper triangular matrix in its left upper corner and the remaining rows of $C$ are zero. The system is solvable if the current right hand side has no nonzero entry in row *rank* and below. Assume otherwise, say $C(i,cols)$ != 0. Then the $i$-th row of $L$ proves the unsolvability of the system.

⟨*test whether a solution exists and compute c if there is no solution*⟩≡
```
for(i = rank; i < rows && C(i,cols) == 0; ++i); // no body
if (i < rows)
{ solvable = false; c = L.row(i); }
```

We compute the solution space. It is determined by a solution $x$ of the non-homogeneous system plus $cols - rank$ linearly independent solutions to the homogeneous system.

Recall that $C$ has a $rank \times rank$ upper triangular matrix in its upper left corner. We view the variables corresponding to the first *rank* columns as dependent and the others as independent. The vector *var* connects columns with variables: column $j$ represents variable $var[j]$.

The components of $x$ are rational numbers with common denominator *denom* (by Cramer's rule and since *denom* is (up to sign) equal to the determinant of the submatrix formed by the dependent variables). We set the components corresponding to the independent variables to zero and compute the components corresponding to the dependent variables by back substitution. During back substitution we compute $x_i$ (again ignoring pivoting) by $x_i = (b_i - \sum_{j>i} c_{i,j}x_j)/c_{i,i}$. Let $x[i]$ be the numerator of $x_i$, then $x[i] = (b_i * denom - \sum_{j>i} c_{i,j} * x[j])/c_{i,i}$.

⟨*compute solution space*⟩≡
```
x = Vector(cols);
D = denom;
for(i = rank - 1; i >= 0; i--) {
  _RT h = C(i,cols) * D;
  for (j = i + 1; j < rank; j++) {
    h -= C(i,j)*x[var[j]];
  }
  x[var[i]]= h / C(i,i);
}
#ifdef CGAL_LA_SELFTEST
  CGAL_assertion( (M*x).is_zero() );
#endif
```

The dimension of the kernel is called the *defect* of the matrix. *spanning_vectors* stores a base of the kernel. We have a spanning vector for each independent variable. We set the value of the independent variable to $1 = denom/denom$ and then compute the values of all dependent variables. In the $l$th spanning vector, $0 \le l < d := defect$ we set variable $var[rank + l]$ to $1 = denom/denom$ and then the dependent variables as dictated by the $rank + l$th column of $C$. The matrix $C$ has the following form

$$\begin{bmatrix} C_{0,0} & \cdots & C_{0,r-1} & C_{0,r} & \cdots & C_{0,n-1} \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & C_{r-1,r-1} & C_{r-1,r} & \cdots & C_{r-1,n-1} \end{bmatrix}$$

For the spanning vector matrix $V$ $(r \times d)$ we set $v_{r+l,l} = 1$ and $v_{r+i,j} = 0$ for $0 \le i < d, 0 \le j < d, i \ne j$. Then we solve the system backwards for a fixed $l$ and all dependent variables :

$$v_{i,l} = (-c_{l,r+l} - \sum_{j>i} c_{i,j}v_{j,l})/c_{i,i}$$

To maintain numerator and denominator separately we have to add a factor $D$ in front of $c_{l,r+l}$ and use only the numerators of the $v_{j,l}$.

⟨*compute solution space*⟩+≡
```
int defect = cols - rank; // dimension of kernel
spanning_vectors = Matrix(cols,defect);
if (defect > 0) {
  for(int l=0; l < defect; l++) {
    spanning_vectors(var[rank + l],l)=D;
    for(i = rank - 1; i >= 0 ; i--) {
      _RT h = - C(i,rank + l)*D;
      for ( j= i + 1; j<rank; j++)
        h -= C(i,j)*spanning_vectors(var[j],l);
      spanning_vectors(var[i],l)= h / C(i,i);
    }
#ifdef CGAL_LA_SELFTEST
      /* we check whether the $l$ - th spanning vector is a solution
         of the homogeneous system */
      CGAL_assertion( (M*spanning_vectors.column(l)).is_zero() );
#endif
  }
}
```

The function determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $\frac{1}{D}x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system. As a precondition we have *M.row_dimension*( ) = *b.dimension*( ).

The presented description of the function *linear_solver* demonstrates the runtime verification underlying our software modules. Either a solution $x$ is calculated which can be easily checked by substitution into the linear system $Mx = b$ or a vector $c$ is provided which proves the unsolvability of the system. Of course there is also a selftest incorporated in the code which can be switched on by a compilation flag and thus the testing can be done permanently.

## 2.3 The Kernel Objects

As shown in the kernel Figure 2.1 we provide the geometric classes *Point_d*, *Vector_d*, *Direction_d*, *Hyperplane_d*, *Segment_d*, *Ray_d*, *Line_d*, *Sphere_d*, and *Aff_transformation_d*. These are the interface types as opposed to the implementation types that specialize the representation. The specialized implementation type for the interface type *Point_d* is the type *PointHd* where the "H" marks the homogeneous representation. There are also implementation types *VectorHd*, *DirectionHd*, *HyperplaneHd*, and *Aff_transformationHd*. We will show how the interface types are mapped to the implementation types later.

We first give a motivation for our interface design and briefly review the basics of analytical geometry. We use $d$ to denote the dimension of the ambient space and assume that our space is equipped with a standard Cartesian coordinate system. The basic object within this space is a point $p$, which we identify with its cartesian coordinate vector $p = (p_0, \ldots, p_{d-1})$, where the $p_i$, $0 \leq i < d$, are rational numbers. The homogeneous point model *PointHd*<*RT*, *LA*> stores homogeneous coordinates $(h_0, \ldots, h_d)$ where $p_i = h_i/h_d$ for all $i$, $0 \leq i < d$, and the $h_i$'s are of ring type *RT*. The homogenizing coordinate $h_d$ is always positive.

Points, vectors, and directions are closely related but nevertheless clearly distinct types. In order to work out the relationship, it is useful to identify a point with an arrow extending from the origin

(= an arbitrary but fixed point) to the point. In this view a point is an arrow attached to the origin. A vector is an arrow that is allowed to float freely in space, more precisely, a vector is an equivalence class of arrows where two arrows are equivalent if one can be moved into the other by a translation of space. Points and vectors can be combined by some arithmetical operations. For two points $p$ and $q$ the difference $p-q$ is a vector (= the equivalence class of arrows containing the arrow extending from $q$ to $p$) and for a point $p$ and a vector $v$, $p+v$ is a point. Compare this layout to the theory of affine spaces in Section 1.1.

All operations of linear algebra apply to vectors, i.e., vectors can be stretched and shrunk (by multiplication with a scalar) and inner and scalar product applies to them. On the other hand, geometric tests like collinearity or orientation only apply to points. Note that we distinguish the vector type *VectorHd* in this scenario of geometric objects from the vector type of the linear algebra module *LA*::*Vector*, which we use to formulate calculations in our arithmetic linear algebra layer. Apart from the conceptual separation in different software modules, their role and thereby their functionality within their respective code module is quite different.

A direction is also an equivalence class of arrows, where two arrows are equivalent if one can be moved into the other by a translation of space followed by stretching or shrinking. Alternatively, we may view a direction as a point on the unit sphere. In two-dimensional space directions correspond to angles. As in the case of *PointHd* we store *VectorHd*, and *DirectionHd*, respectively, as a homogeneous tuple of integers with positive homogenizing component.

The common one-dimensional straight-line objects in $d$-space like lines, rays and segments (which we allow to be trivial) are implemented in the classes *Line_d*, *Ray_d* and *Segment_d* and are determined by a pair of points.

With respect to the user interface we can group together points, vectors, directions, and on the other hand segments, rays, and lines. For the first group there are common operations to access Cartesian and homogeneous coordinates. Conversions within the first group can be made by explicit operations. The second group can be seen as container classes determined by a pair of points. As we will see later this makes it possible to save on a specialization with respect to coordinate representation (there are no types *LineHd*, *RayHd*, or *SegmentHd*.).

Oriented hyperplanes in the class *HyperplaneHd* can be used to model half-spaces and affine hulls of $(d-1)$-dimensional point sets. They are internally stored as a $(d+1)$-tuple of *RT* coefficients.

Finally oriented spheres of type *Sphere_d* are helpfull in proximity calculations like Voronoi diagrams or Delaunay triangulations. They are stored as a tuple of $d+1$ points (type *Point_d*) and they are also not specialized via representation.

For all of our basic geometric types we have affine transformations, which can be used by a call of a common member operation which gets an *Aff_transformationHd*-instance as an argument and delivers a transformed object.

All object classes mentioned use a smart pointer scheme which is already used in many modules of LEDA and CGAL (see page 14). We distinguish between a front-end object which is created by the constructor and a storage object of concrete geometric information which is referenced from the front-end object. The advantages of this scheme emerge in case of frequent copy construction and assignment where only references have to be redirected and no geometric information has to be copied. Whenever an object is copy constructed or assigned only an additional front-end object is created whose reference is set to the background storage object (whose reference counter is increased). Destruction of a front-end object decreases the counter. A background object is destroyed whenever the counter falls to zero.

For large objects this decreases memory consumption and allows us to improve equality checks by testing the reference addresses before a comparison of geometric coordinate information. Both the

linear algebra module and the geometric objects are parameterized by a memory allocation scheme. CGAL uses LEDA's improved memory management module when both libaries are installed. This gives us a certain speed-up compared to the standard C++ allocation scheme. All objects of the kernel are programmed along this smart pointer scheme.

We now take a closer look at the interface properties as defined by the manual pages, by which we further elaborate on the features of our data types. The Appendix 4.1 shows the corresponding manual pages.

**Points, Vectors and Directions** points, vectors, and directions in $d$-dimensional space are defined by their coefficients. A corresponding object is *not* mutable. One cannot change one coefficient but keep the rest. This provision is necessary to make the smart pointer scheme efficient. The interface now specifies the ways how the coefficient tuple can be transferred into the objects and how they can be read from the outside.

The generic way to specify a sequence of objects is an iterator range. Therefore the construction of a point with respect to an iterator range $[start, end)$ referencing a tuple of $RT$ coordinates is either $p(d, start, end, D)$ or $p(d, start, end)$ where in the first version the range specifies $d$ numerator values of the coefficients and $D$ is the common denominator. In the second version the denominator is the last entry of the iterator range that has to contain $d+1$ values.

The data access operations allow access to the dimension via $p.dimension(\,)$ (the ambient dimension $d$). The $i$th Cartesian and homogeneous coordinate is accessible via $p.cartesian(i)$ and $p.homogeneous(i)$ where the first operation returns the quotient $p.homogeneous(i)/p.homogeneous(d)$ of $RT$ values and the second directly the corresponding $RT$ entry. Additionally, there are also read-only iterator ranges for simple output iteration via the method pairs *homogeneous_begin*( )/*_end*( ) and *cartesian_begin*( )/*_end*( ). Operator overloading allows the intuitive calculation of *Point_d* difference, which results in a *Vector_d* and the translation of *Point_d* by adding a *Vector_d*.

Vectors have a similar interface to points. Only that the applicable operations are those of linear algebra. Thus a vector $v$ can be multiplied and divided by scalars of type $RT$ and $FT$, two vectors $v$ and $w$ can be added $v+w$ or subtracted $v-w$. One can also apply the standard Euclidean inner product $v*w$ and determine the squared length $v.squared\_length(\,)$.

Points and Vectors can be converted to each other by an overloaded addition and substraction involving a global CGAL object called *ORIGIN*. For a vector $v$ the sum $ORIGIN+v$ determines a point object of corresponding coordinates. Reversely, $p-ORIGIN$ returns the vector that corresponds to $p$.

Finally, a vector $v$ can be converted to a direction by a call to $v.direction(\,)$ and the converse operation is provided by $d.vector(\,)$. Directions offer simple operations reflecting their special character as vectors where we forget about their length.

**Segments, Rays, and Lines** To represent the group of straight line objects we appended the manual page of *Line_d* in the appendix. The basic operations provided on these objects are mainly their construction from other kernel objects, like $l(p,q)$ constructs the line $l$ through two points $p$, $q$, access to their defining points $l.point(i)$ for $i = 0, 1$, and position checks with respect to the objects, like $l.has\_on(p)$ or $parallel(l1, l2)$ where *l1* and *l2* are two lines. Only the latter require calculations with respect to the points that span the line.

**Spheres** An object $S$ of type *Sphere_d* is an oriented sphere of $d$-space. A sphere is defined by $d+1$ points with rational coordinates (class *Point_d*). We use $A$ to denote the tuple of the defining

points. A set $A$ of defining points is *legal* if either the points are affinely independent or if the points are all equal. Only a legal set of points defines a sphere in the geometric sense and hence many operations on spheres require the set of defining points to be legal. The orientation of $S$ is equal to the orientation of the defining points in $A$. Spheres can be constructed from an iterator range of points. One can query equality, non-equality, its center and squared radius, and if some point of type *Point_d* is contained in the bounded interior, in the boundary or unbounded outer region.

**Predicates and Constructions** Apart from the method interface of the geometric objects, the main value of the kernel are its predicates and constructions including intersection calculations. We sketch some functionality. The main objects are points and vectors and tuples thereof. We again adopt their generic abstraction and use iterator ranges $[start, end)$ as arguments of the primitives. In the affine space we provide predicates like *affinely_independent*$(start, end)$, *contained_in_simplex*$(start, end, p)$ or *orientation*$(start, end)$ where *tuple* $[start, end)$ is a tuple of points. The first returns *true* iff the tuple of points is affinely independent, the second returns true iff $p$ is contained in the simplex spanned by the points in $tuple$ $[start, end)$, and the latter returns the *Orientation*[5] of *tuple* $[start, end)$ where *size* $[start, end)$ must be $d+1$. For explorations of linear spaces there are also predicates defined on vectors. For example, *linear_rank*$(start, end)$ returns the dimension of the linear space spanned by vectors in *set* $[start, end)$. A complete list of predicates can be found in the appendix. We will describe implementation issues below.

**Intersections** The intersection interface follows the polymorphic design of the lower dimensional kernel. For each pair of objects *o1, o2* of the types *Line_d, Ray_d, Segment_d* and *Hyperplane_d* there is an intersection operation *intersect*$(o1, o2)$ that returns a polymorphic object $o$ of type *Object*. This object can be typewise identified and assigned depending on the possible results of the intersection operation by the function *bool assign*$(T, Object)$ (where $T$ is any type to which we want to assign). To make the interface clear we present an example. Assume we intersect, a ray $r$ and a hyperplane $h$. The result can be a common point $p$, the ray $r$, or no intersection. Then the intersection can be calculated by

```
Object i = intersect(h,r);
if ( assign(p,i) ) ...      // do something with point p
else if ( assign(r,i) ) ... // do something with ray r
else ...                    // no intersection
```

the *assign* operation returns *true* iff the polymorphic conversion from $i$ to the corresponding object (first parameter) can be done. Note that this scheme requires the user to know about the possible outcome of the intersection operation to query the corresponding result type.

If we only want to check for intersections there is also a pairwise operation *do_intersect*$(o1, o2)$ that returns *true* iff the objects intersect.

**Affine Transformations** As introduced in the introduction, any transformation of the form $\mathfrak{y} = \mathbf{A}\mathfrak{x} + \mathfrak{b}$ is an affine transformation of a point $\mathfrak{x} = (x_0, \ldots, x_{d-1})$. Due to the fact that our representation is in homogeneous coordinates $h_0, \ldots, h_d$ there is a more compact way to represent such transformations by $d+1$-dimensional square matrices. Note that we require the entries of $\mathbf{A}$ and $\mathfrak{b}$ to be all rational numbers. Let $w$ be the least common multiple of all the denominators of $\mathbf{A}$ and $\mathfrak{b}$. We can then represent the components of the affine transformation by an

[5] *Orientation* is an enumeration type of CGAL with constants *POSITIVE, ZERO, NEGATIVE*.

integral matrix $\mathbf{A}'$ and an integral $\mathfrak{b}'$ and the common denominator $w$ such that $a'_{ij}/w = a_{ij}$ and $b'_i/w = b_i$. Note that the following $(d+1)\times(d+1)$ matrix $\mathbf{M}$ can be used to transform the homogeneous representation of $\mathfrak{x} = (h_0,\ldots,h_d)$ by simple matrix multiplication:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A}' & \mathfrak{b}' \\ 0 & w \end{bmatrix}$$

as

$$\mathbf{M}\mathfrak{x} = \begin{cases} \sum_{j=0}^{d-1} a'_{ij}h_j + b'_i h_d & 0 \le i < d \\ wh_d & i = d \end{cases}$$

We obtain the standard Cartesian transformation by dividing the first row (the components $i = 0,\ldots,d-1$) by the second row. Our affine transformation data type stores the matrix $\mathbf{M}$ via a smart pointer scheme and offers diverse construction mechanism for standard transformations of type *Aff_transformation_d<R>*:

$t(M)$ constructs the transformation according to a matrix $\mathbf{M}$, $t(start,end)$ constructs the componentswise scaling transformation according to the iterator range of $d+1$ *RT* values, $t(d,num,den)$ constructs a uniform scaling of dimension $d$ with scaling factor $num/den$, $t(v)$ constructs the translation as defined by a vector $v$ in $d$-space of type *Vector_d<R>*.

There is also a construction scheme to obtain approximate rotations as specified by rational directions specified in a linear subspace of dimension 2 and determinded by two coordinate axes. $t(d,dir,num,den,i,j)$ constructs a rotation in the subspace spanned by the coordinate axes $i$ and $j$ $(0 \le i < j < d)$ of $d$-space and approximating the angle specified by *dir*. The approximation is such that the difference between the sines and cosines of the rotation given by *dir* and the entries of the rotation matrix are at most $num/den$. The code is based on the rational rotation method presented by J. Canny and E.K. Ressler [CDR92].

## 2.4 Applications

The convex hull and the Delaunay triangulation problem are traditionally specified as functions, i.e., given a set of points, compute their convex hull or their Delaunay triangulation in some representation. We specify both problems as data types that support insertions and a large variety of query operations. In the case of convex hulls we support navigation through the interior and the boundary of the hull and we support membership and visibility queries. In the case of Delaunay triangulations we support navigation through the triangulation, we support locate[6] and nearest neighbor queries, and we support range queries with spheres and simplices. For two-dimensional convex hulls and Delaunay triangulations we also support an interface to the LEDA graph and window classes [MNSU99, MN99], as well as to CGAL polyhedral surfaces. In this way one can, for example, construct two-dimensional nearest and furthest site Voronoi diagrams and minimum spanning trees, display hulls and Delaunay triangulations.

The next two sections present parts of the specifications of convex hulls and Delaunay triangulations, respectively.

**Convex hulls** — An instance $C$ of type *Convex_hull_d<R>* is the convex hull of a multi-set $S$ of points in $d$-dimensional space. We call $S$ the underlying point set and $d$ or *dim* the dimension of the

[6] A locate-query finds the simplex of the triangulation containing the query point

underlying space. We use *dcur* to denote the affine dimension of $S$. The data type supports incremental construction of hulls.

The closure of the hull is maintained as a simplicial complex, i.e., as a collection of simplices the intersection of any two is a face of both[7]. In the sequel we reserve the word simplex for the simplices of dimension *dcur*. For each simplex there is a handle of type *Simplex_handle* and for each vertex there is a handle of type *Vertex_handle*. Each simplex has $1+dcur$ vertices indexed from 0 to *dcur*; for a simplex $s$ and an index $i$, $C.vertex(s,i)$ returns the $i$-th vertex of $s$. For any simplex $s$ and any index $i$ of $s$ there may or may not be a simplex $t$ opposite to the $i$-th vertex of $s$. The function $C.opposite\_simplex(s,i)$ returns $t$ if it exists and returns *Simplex_handle*( ) (the undefined handle) otherwise. If $t$ exists then $s$ and $t$ share *dcur* vertices, namely all but the vertex with index $i$ of $s$ and the vertex with index $C.index\_of\_vertex\_in\_opposite\_simplex(s,i)$ of $t$. Assume that $t$ exists and let $j = C.index\_of\_vertex\_in\_opposite\_simplex(s,i)$. Then $s = C.opposite\_simplex(t,j)$ and $i = C.index\_of\_vertex\_in\_opposite\_simplex(t,j)$.

Again we appended the specification of the convex hull data type in the appendix.

**Delaunay triangulations** — An instance *DT* of type *Delaunay_d<R,Lifted_R>* is the nearest and furthest site Delaunay triangulation of a set $S$ of points in some $d$-dimensional space. We call $S$ the underlying point set and $d$ or *dim* the dimension of the underlying space. We use *dcur* to denote the affine dimension of $S$. The data type supports incremental construction of Delaunay triangulations and various kind of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

A Delaunay triangulation is a simplicial complex. All simplices in the Delaunay triangulation have dimension *dcur*. In the nearest site Delaunay triangulation the circumsphere of any simplex in the triangulation contains no point of $S$ in its interior. In the furthest site Delaunay triangulation the circumsphere of any simplex contains no point of $S$ in its exterior. If the points in $S$ are co-circular then any triangulation of $S$ is a nearest as well as a furthest site Delaunay triangulation of $S$. If the points in $S$ are not co-circular then no simplex can be a simplex of both triangulations. Accordingly, we view *DT* as either one or two collection(s) of simplices. If the points in $S$ are co-circular there is just one collection: the set of simplices of some triangulation. If the points in $S$ are not co-circular there are two collections. One collection consists of the simplices of a nearest site Delaunay triangulation and the other collection consists of the simplices of a furthest site Delaunay triangulation.

For each simplex of maximal dimension there is a handle of type *Simplex_handle* and for each vertex of the triangulation there is a handle of type *Vertex_handle*. Each simplex has $1+dcur$ vertices indexed from 0 to *dcur*. For any simplex $s$ and any index $i$, $DT.vertex\_of(s,i)$ returns the $i$-th vertex of $s$. There may or may not be a simplex $t$ opposite to the vertex of $s$ with index $i$. The function $DT.opposite\_simplex(s,i)$ returns $t$ if it exists and returns *nil* otherwise. If $t$ exists then $s$ and $t$ share *dcur* vertices, namely all but the vertex with index $i$ of $s$ and the vertex with index $DT.index\_of\_vertex\_in\_opposite\_simplex(s,i)$ of $t$. Assume that $t = DT.opposite\_simplex(s,i)$ exists and let $j = DT.index\_of\_vertex\_in\_opposite\_simplex(s,i)$. Then $s = DT.opposite\_simplex(t,j)$ and $i = DT.index\_of\_vertex\_in\_opposite\_simplex(t,j)$. In general, a vertex belongs to many simplices.

Any simplex of *DT* belongs either to the nearest or to the furthest site Delaunay triangulation or both. The test $DT.simplex\_of\_nearest(Simplex\_handle\ s)$ returns true if $s$ belongs to the nearest

[7]The empty set if a facet of every simplex.

site triangulation and the test *DT.simplex_of_furthest*(*Simplex_handle s*) returns true if *s* belongs to the furthest site triangulation.

**Further implementation issues** — The implementation of type *Convex_hull_d* follows [CMS93] and Delaunay triangulations are reduced to convex hulls through the well-known lifting map, see for example [Ede87]. Based on our kernel, a class *Regular_complex_d* was implemented that can represent so-called regular simplicial complexes. A simplicial complex is called regular if all maximal simplices, i.e., simplices that are not a subsimplex of another simplex of the complex, have the same dimension. The class *Regular_complex_d* provides operations for navigation through the complex and update operations. The class *Convex_hull_d* is derived from *Regular_complex_d* and the class *Delaunay_d* is derived from *Convex_hull_d*.

The work horse for the query operations on convex hulls and Delaunay triangulations is a method

```
C.visibility_search(Point_d x, std::list<Facet_handle>& visible_facets,
                    int& location, Facet_handle& f);
```

that constructs the list of all *x*-visible hull facets in *visible_facets*, returns the position of *x* with respect to the current hull in *location* ($-1$ for inside, 0 for on the the boundary, and $+1$ for outside) and, if *x* is contained in the boundary of *C*, returns a facet incident to *x* in $f$.

The membership query and the visible facets query for hulls are easily realized by this method and the nearest neighbor and the range query for Delaunay triangulations use it in an essential way. The nearest neighbor query for Delaunay triangulations lifts the query point (using the lifting map), then determines all visible facets of the hull, and then selects the best vertex by linear search through their vertices[8]. The range query with spheres lifts the sphere (using the lifting map) and then finds all vertices of the hull that lie below the resulting hyperplane[9].

In order to make the results of our applications verifiable, we use program checking [BLR90, MNS+96] in our implementation. In particular,

- the class *Regular_complex_d* provides a method *RC.check_topology*( ) that partially checks whether *RC* is an abstract simplicial complex[10].
- the class *Convex_hull_d* provides a method *is_valid*( ) that verifies convex hulls as described in [MNS+96].

The representation of convex hulls and Delaunay triangulations in data types *Convec_hull_d* and *Delaunay_d* is simplex-based, i.e., simplices are the main objects and lower dimensional faces are only implicitly represented. In lower-dimensional space there are alternative representations. Two dimensional convex hulls and Delaunay triangulations can be represented by planar graphs: the vertices and edges are the primary objects and represent simplices (= triangles) implicitly as faces of the planar graph. For convex hulls in 3-space, there is also a function that extracts the surface of a convex hull and returns a polyhedral surface.

If *C* has type *Convex_hull_d*<*R*>, *P* has type *Polyhedron_3*<*T*>, where the point type *Polyhedron_3*<*T*>:: *Point* (introduced by the traits class *T*) has to be the same as

[8] This method is only efficient in low-dimensional space.

[9] The lifting map turns a sphere into a hyperplane.

[10] The method checks whether the neighborhood relationship on simplices is symmetric, whether all vertices of a simplex are distinct, and whether two neighboring simplices share all but one of their vertices. It does not check whether simplices that share all but one of their vertices are actually neighbors in the complex.

*Convex_hull_d*<*R*>::*Point_d* then *convex_hull_d_to_polyhedron_3* $(C, P)$ extracts the surface of $C$ and stores it in $P$.

## 2.5 Generic Programming Techniques

Linear algebra and the number type are conceptually introduced as template parameters of *Homogeneous_d*. The *LA* parameter defaults to our default model *Linear_algebraHd*<*RT*>.

```
template <class RT_, class LA_ = Linear_algebraHd<RT> >
struct Homogeneous_d {
  typedef RT_ RT;
  typedef LA_ LA;
  typedef Quotient<RT> FT;...
```

Note that we provide the arithmetic types as local types of the kernel. *RT* is called the ring type, *FT* is the corresponding field type, and *LA* the linear algebra type. The above type redirection is necessary to make *RT* and *LA* available as member types of *Homogeneous_d*. From now on we will save on that pattern and implicitly use *RT* and *LA* in the class scope. Sometimes we will just write *Homogeneous_d*<*RT*> instead of *Homogeneous_d*<*RT*,*LA*>.

A kernel is a concept and therefore carries types and functionality that it offers to a user or that can be used by application programs when the representation class is used as a traits class. There are interface types and internally used types. We want to use global types that are *parameterized* by the representation class but that are at the same time contained in the traits class to make them available for application programs. C++ supports the following scheme by allowing a delayed instantiation.

```
template <class RT, class LA>
struct Homogeneous_d {
 typedef PointHd<RT,LA> Point_d_base;
 ...
template <class R>
class Point_d : public R::Point_d_base {
 ...
```

Up to this point there is nothing new. But we can just make *Point_d*<*R*> a member of *Homogeneous_d* and thereby using the latter in two roles: once as a carrier for the base class *Point_d_base* and once as a container of the interface type *Point_d* that is derived from the base class.

```
template <class RT, class LA>
struct Homogeneous_d {
 typedef Homogeneous_d<RT,LA> Self;
 typedef CGAL::PointHd<RT,LA> Point_d_base;
 typedef CGAL::Point_d<Self>  Point_d;
 ...
```

Note that now *Point_d*<*R*> is equal to *R*::*Point_d* (where *R* is a model of the kernel, *e.g.*, *Homogeneous_d*<*RT*>). The above trick is also used for the kernel types *Vector_d*, *Direction_d*, and *Hyperplane_d*.

There is also a second type of objects in the kernel. Affine objects representing point sets that are not strongly related to the representation of the coordinates. We coded them in a meta layer such that the classes work for both the homogeneous and the Cartesian representation classes. The classes are *Segment_d*, *Ray_d*, *Line_d*, and *Sphere_d*.

```
template <class R>
class Segment_d {
  typedef typename R::Point_d Point_d;
  // make Segment_d a smart pointer to a pair of Point_d objects
 ...
template <class RT, class LA>
struct Homogeneous_d {
 typedef Homogeneous_d<RT,LA> Self;
 typedef CGAL::Segment_d<Self> Segment_d;
 ...
```

All interface operations that do algebraic calculations (there are only few, like *Segment_d<R>*:: *has_on*(*Point_d x*)) get their functionality from predicate units that are placed in the representation class *R* and are therefore specialized with respect to the representation class *R*.

### Predicates and Constructions

The role of the kernel as a traits class requires carrying predicates and constructions that are needed by application programs. Function objects are the predicate and construction units. We show how we implement two predicates and how the interface accesses the implementation.

Our first example is the predicate that determines whether a point $p$ is contained in the affine hull of a set of points $A = set[first, last)$ of an iterator range.

```
template <class R>
struct Contained_in_affine_hullHd {
typedef typename R::Point_d Point_d;
typedef typename R::LA LA;
template <class ForwardIterator>
bool operator()(ForwardIterator first, ForwardIterator last,
                const Point_d& p)
{
  TUPLE_DIM_CHECK(first,last,Contained_in_affine_hull_d);
  int k = std::distance(first,last); // |A| contains |k| points
  int d = first->dimension();
  typename LA::Matrix M(d + 1,k);
  typename LA::Vector b(p.homogeneous_begin(),p.homogeneous_end());
  for (int j = 0; j < k; ++first, ++j)
    for (int i = 0; i <= d; ++i)
      M(i,j) = first->homogeneous(i);
  return LA::is_solvable(M,b);
}
};
```

A point $p$ is contained in the affine hull of a set $A$ of points if $p$ is an affine combination of the points in $A$, i.e., if the system $\sum \lambda_i A_i = p$ has a solution with $\sum \lambda_i = 1$. Set $\lambda_i = A_{i,d}\beta_i / p_d$ with $A_{i,d}$ being the homogenizing coordinate of $A_i$ and $p_d$ being the homogenizing coordinate of $p$. The $i$-th column of the system for the $\beta_i$'s is simply the homogeneous vector of $A_i$ and the right hand side is simply the homogeneous vector for $p$.

The above function object is provided in the kernel as a member type. The specialized and representation dependent function object type *Contained_in_affine_hullHd<Self>* is mapped to the member type *Contained_in_affine_hull_d* (one concept requirement of the kernel).

```
template <class RT, class LA>
struct Homogeneous_d {
  typedef Homogeneous_d<RT,LA> Self;
  typedef Contained_in_affine_hullHd<Self> Contained_in_affine_hull_d;
  Contained_in_affine_hull_d contained_in_affine_hull_d_object() const
  { return Contained_in_affine_hull_d(); }
  ...
```

There are two interfaces to use the function object. The first is the usage of the predicate function object directly when *Homogeneous_d<RT>* is used as traits class. Note that traits classes dock to applications not only as types but also as objects that might carry runtime information. Therefore using traits classes comprehend the forwarding of function objects (that implement predicates and constructions). In this way the function objects that a traits class forwards to an application can bind runtime information that is part of its calculations. (*e.g.*, imagine an orientation predicate in a plane where the plane is fixed within the traits class). The interface operation to obtain the function object is *contained_in_affine_hull_d_object*( )[11] in the above case. Note that this operation just uses the default constructor of the function object, however the *_object*( )-function is there because *Homogeneous_d<RT>* is a model of the traits concept.

For the global kernel view we offer a simple global predicate function that makes sense if a user does not use the traits mechanism but is just using the kernel from its global abstract interface. A global function like *contained_in_affine_hull* can thus rely on the function object type from the kernel (it works also for different representation types like a *Cartesian_d* kernel model).

```
template <class R, class ForwardIterator>
bool contained_in_affine_hull(ForwardIterator first, ForwardIterator last,
                              const Point_d<R>& p)
{ typename R::Contained_in_affine_hull_d contained;
  return contained(first,last,p); }
```

Our second example is the orientation predicate. We cite from the manual specification. The predicate determines the orientation of the points in the set $A = set\,[first, last)$ where $A$ consists of $d+1$ points in $d$ - space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \dots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the $i$-th point in $A$. One precondition is that $size\,[first, last) == d+1$ and $A[i].dimension(\,) == d\ \forall 0 \le i \le d$. The value type of *ForwardIterator* has to be *Point_d<R>*.

```
template <class R> struct OrientationHd {
typedef typename R::Point_d Point_d;
typedef typename R::LA LA;

template <class ForwardIterator>
Orientation operator()(ForwardIterator first, ForwardIterator last)
{ TUPLE_DIM_CHECK(first,last,Orientation_d);
  int d = std::distance(first,last);
  // range contains d points of dimension d-1
  CGAL_assertion_msg(first->dimension() == d-1,
  "Orientation_d: needs first->dimension() + 1 many points.");
  typename LA::Matrix M(d); // squared matrix
```

[11] CGAL has the naming convention to write the object name lowercase and append the *_object*( ) postfix.

```
  for (int i = 0; i < d; ++first,++i) {
    for (int j = 0; j < d; ++j)
      M(i,j) = first->homogeneous(j);
  }
  int row_correction = ( (d % 2 == 0) ? -1 : +1 );
  // we invert the sign if the row number is even i.e. d is odd
  return Orientation(row_correction * LA::sign_of_determinant(M));
}
};
```

The implementation is a straightforward translation of the above by means of the linear algebra carried in the kernel. The code excerpt shows the vertical interaction between the software layers. The macro *TUPLE_DIM_CHECK* checks that all objects of the iterator range have the same dimension. The function *std*::*distance*( ) determines *size* $[first, last)$ depending on the iterator category. We allocate the square matrix that is filled by the homogeneous point coefficients. As our coefficient vector layout has the homogenizing coordinate in the last position we add the *row_correction* to account for the row flipping in different dimensions. Finally the *sign_of_determinant* operation is the working unit of the predicate. Most affine operations on point tuples map to the solution of a corresponding linear system calculated by our linear algebra layer.

The above function object is again provided in the kernel as a member type. The specialized and representation dependent function object type *OrientationHd*<*Self*> is mapped to the member type *Orientation_d*.

```
template <class RT, class LA>
struct Homogeneous_d {
  typedef Homogeneous_d<RT,LA> Self;
  typedef OrientationHd<Self> Orientation_d;
  Orientation_d orientation_d_object() const
  { return Orientation_d(); }
  ...
```

Again there are the two interfaces to use the function object. The first is the *orientation_d_object*( ) method that again makes *Homogeneous_d*<*RT*, *LA*> a model of the kernel concept. For the global kernel view we offer again a simple global predicate function. In this case the parameter list does not contain a type reference to the kernel type. For global functions based solely on iterator ranges we obtain the kernel via the *iterator_traits* mechanism.

```
template <class ForwardIterator>
Orientation
orientation(ForwardIterator first, ForwardIterator last)
{
  typedef typename std::iterator_traits<ForwardIterator>::value_type Point_d;
  typedef typename Point_d::R R;
  typename R::Orientation_d orientation_;
  return orientation_(first,last);
}
```

The iterator traits exhibits the value type of the iterator. From this value type we obtain the kernel and finally the orientation function object type.

### Application traits classes

As already described in Section 2.2, the canonical way of parameterization of an application is via a single parameter that corresponds to a model of the kernel (as a traits class). We follow this scheme in the design of *Regular_complex_d<R>* and *Convex_hull_d<R>*. A flexible design of *Delaunay_d<R, Lifted_R>* requires the separation of two traits concepts. The first parameter *R* refers to the ambient $d$-space of the Delaunay triangulation, and the second refers to the lifted $d+1$-space. Our default model *Homogeneous_d<RT>* fits both parameters and actually the second parameter defaults to the first, such that *Delaunay_d<R>* $=$ *Delaunay_d<R, R>*. The decoupling is necessary in cases where the geometry in dimension $d+1$ is implemented by a different kernel than that of the ambient space. A trivial example is the specialization of the $d$-dimensional algorithm to 2-dimensional delaunay triangulations. There the traits models for dimension 2 and 3 can be based on the available kernels for the fixed dimension.

The traits requirements of *Convex_hull_d<R>* and *Delaunay_d<R, Lifted_R>* are documented at the end of their manual pages. Note that an application object obtains a kernel object on construction. This is necessary as a kernel model might carry runtime information into the application. Convex hull construction follows the following prototype

```
Convex_hull_d(int d, const R& Kernel = R());
```

The corresponding kernel reference *Kernel* is stored in the convex hull object and can be accessed by a call to the *kernel*( ) method.

We give an example how the application class interacts with the kernel as a traits model. The following operation is taken from the implementation of class *Convex_hull_d<R>*. It implements the check if some point *x* causes a dimension jump of the hull when inserted into the current convex hull object.

```
bool is_dimension_jump(const Point_d& x) const
{
  if (current_dimension() == dimension()) return false;
  typename R::Contained_in_affine_hull_d contained_in_affine_hull =
    kernel().contained_in_affine_hull_d_object();
  return ( !contained_in_affine_hull(origin_simplex_->points_begin(),
    origin_simplex_->points_begin()+current_dimension(),x) );
}
```

## 2.6 Discussion and Experiments

The full documentation of the application layer comprises about 100 pages[12]. For the application programs and the linear algebra module the documentation and the implementation are collected in a noweb-file[13] and different tools are used to give different views of the noweb-file: the noweb tool `notangle` extracts the code, i.e., the view needed by the C++ compiler, and the LEDA tools `Lman` and `Ldoc` give the manual view and the documentation view, respectively[14].

Both application programs are only reliable if the kernel is exact. For example, the insertion routine for convex hulls distinguishes cases according to whether the newly inserted point lies in the

[12] see `http://www.mpi-sb.mpg.de/LEDA/friends/dd_geokernel.html`

[13] see `http://www.cs.purdue.edu/homes/nr/noweb` for an introduction to noweb.

[14] see the LEDA book [MN99] for an introduction to these tools

Table 2.1: *Convex_hull_d*<...> varying dimensions, number types, and representations

| dimension | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| vertices | 34 | 49 | 69 | 91 | 91 | 98 | 100 |
| facets | 65 | 218 | 1035 | 4944 | 16233 | 63958 | 231473 |
| double cartesian | 0.01 | 0.03 | 0.13 | 0.84 | 3.44 | 17.33 | 79.77 |
| double homogeneous | 0 | 0.02 | 0.12 | 0.78 | 3.37 | 17.96 | 85.32 |
| LEDA integer homogeneous | 0.03 | 0.1 | 0.69 | 4.98 | 24.34 | 138.93 | 716.78 |
| LEDA real cartesian | 0.04 | 0.12 | 0.84 | 5.15 | 23.6 | 124.02 | - |
| GNU mpz homogeneous | 0.06 | 0.27 | 1.79 | 13.04 | 63.91 | 359.93 | 1992.28 |

`chull_dd-runtime 15` on inputs created by `chull_dd-runtime 0 d 100`

affine hull of the points already present or not, and the checking programs does hardly make sense without exact primitives.

In the early stages of program development the checking feature of the kernel was particulary useful. For example, the convex hull program needs to compute the hyperplane defined by a set of points. This can be done by solving a linear system. In the first version of the program we set up the wrong linear system. It was very useful that the linear system solver gives a proof of unsolvability and does not just claim unsolvability. This located the error fairly quickly.

We have used classes *Convex_hull_d* and *Delaunay_d* on problems up to dimension 10. We have also compared it to the *qhull*-program of Barber, Dobkin, and Hudhanpaa [BDH96] and the *hull*-program of Clarkson. The first method computes approximate convex hulls and the latter method computes exact hulls but works only for a limited (albeit large) range of coordinate values. Both methods are significantly faster than ours. This is mostly due to their use of floating point arithmetic. However, neither of the algorithms provides the rich functionality that we provide. Note that *Convex_hull_d* provides an online setting whereas *quull* and *hull* work in a setting where the input is fixed.

We first instantiate *Convex_hull_d*<> with different kernels. The runtime results are shown in Table 2.1. We produced a sample of 100 random points in the $d$-dimensional cube of size 100. Then we instantiated the convex hull data type with the two kernel representation families with different number types. We took the running times for inserting the 100 points into an empty complex. The table shows the number of vertices and facets per dimension and the time to construct the simplicial complex in seconds. The program `chull_dd-runtime.C` can be found in the CGAL source tree in `CGALROOT/demo/Convex_hull_d`. It was compiled with `g++ -O2` and run on a SUN Sparc E10000 with huge core memory. Running `chull_dd-runtime 0 dim n` produces an input file `chull_dd-runtime.ch` that is in *qhull* input format. Then this input can be used with `chull_dd-runtime choice` where *choice* is numeric flag that allows to choose the corresponding instantiation of *Convex_hull_d* (*e.g.*,`chull_dd-runtime 15` chooses all the instantiation possible). `chull_dd-runtime -h` gives an overview of the corresponding options.

In our small number range the standard double instantiations (homogeneous and Cartesian) worked well when the consistency tests of the program where switched off. The LEDA number type *real* worked also well up to a certain dimension. However its space requirements made it fail starting from dimension 8. The instantiation of choice is the homogeneous kernel with LEDA *integer*s. The main bottleneck of *Convex_hull_d* is its memory consumption for the large number of simplices built in higher dimensions. We give a comparison of the runtimes and complexity of *Convex_hull_d* and

Table 2.2: Comparison *Convex_hull_d*< *Cartesian_d*<*double*> > and *qhull*

| number of points | 200 | 400 | 600 | 800 | 1000 |
|---|---|---|---|---|---|
| Convex_hull_d | 1.91 | 3.92 | 5.78 | 8.3 | 8.38 |
| Vertices/Facets | 155/9970 | 279/19951 | 348/26444 | 442/34886 | 438/35588 |
| qhull | 0.15 | 0.3 | 0.53 | 0.63 | 0.63 |
| Vertices/Facets | 126/1986 | 208/3848 | 250/4665 | 292/5495 | 308/5899 |

`chull_dd-runtime 1` and `qhull` on inputs created by `chull_dd-runtime 0 5 n`

*qhull* in Table 2.2. We compare the Cartesian double version of *Convex_hull_d* with *qhull* in dimension 5. The table presents a lower bound of *Convex_hull_d* runtime. The runtimes show that the online insertion scheme of *Convex_hull_d* is inherently slower than the heuristic input filtering of the quick hull algorithm when used in a static input scenario. Starting from 600 input points our checker reported local non-convexities at ridges of the convex hull. However the program did not crash. Note that the *qhull* counts only extreme vertices and non-simplicial facets, whereas the *Convex_hull_d* statistics are the boundary vertices and facets of the simplicial complex.

At the beginning of this project we also compared the pure linear algebra module to the standard math packages Maple [CGG+91a] and Mathematica [Wol96]. Our implementation is significantly faster. Please refer to the corresponding technical report [See96]. We also tested alternative approaches for the homogeneous linear algebra module sketched above. In his master thesis, O. Ashoff [Ash99] implemented a model of our LA-concept based on a description of M. McClellan [M.T73]. The linear algebra model was faster than ours in high dimensions with large numbers. However, the instance did not show runtime improvements for the calculation of convex hulls but was actually slower. For instances up to dimension 10, the homogeneous linear solver seems to be well tuned.

## 2.7 Conclusions

We implemented and tested the usability of a generic higher-dimensional kernel in CGAL. Its main features are the exchangeability of number types, representation types and linear algebra and the built-in checking features. Our applications follow the exact arithmetic paradigm and by using the corresponding instantiation we can handle all degenerate cases and the computed results are fully checkable. One interesting extension would be the implemenation of the original quick hull algorithm on top of our geometric kernel and to compare this to the monolithic design of the original implementation. The runtime tests in the previous sections show the main advantages of our generic design. The possible replacement of number types and linear algebra modules is one of its major strengths compared to a monolithic design.

The price for the generic design of the kernel was certainly a longer development cycle. The implementation of template code enforces a strict discipline with respect to testing but our development was also obstructed by technological problems. One target of CGAL was portability, but many state-of-the-art compilers are still incomplete in their realization of the C++ language standard. The usage of cutting-edge technology bears the need for work-arounds or even a voluntary restriction to available functionality for certain platforms. The result however should convince our readers that the obtained flexibility is worth the price.

# Chapter 3

# Infi maximal Frames

## 3.1 Introduction

Many geometric algorithms that are usually formulated for points and segments generalize nicely to inputs containing also rays and lines. Do implementations generalize as easily? Let us consider two concrete examples: plane sweep for segment intersection and map overlay.

In the plane sweep algorithm for segment intersection a vertical line is swept across the plane from left to right. The intersections between the sweep line and the input segments are kept in a data structure, the Y-structure. The Y-structure is updated whenever the sweep line encounters a segment endpoint or an intersection point between two segments. The event points are kept in a priority queue, the X-structure. The sweep paradigm can clearly also handle rays and lines. Will an implementation generalize easily, *e.g.*, does LEDA's implementation [MN99, Section 10.7] generalize? It does not. For example, the X-structure needs to be initialized with the endpoints of the segments, but what are the endpoints of lines and segments?

In Section 3.2 we will argue that the answer is not given by projective geometry (neither standard nor oriented). We will also argue that enclosing the scene in a fixed geometric frame and clipping rays and lines at the frame is an unsatisfactory solution. It excludes on-line algorithms, it requires non-trivial changes in the software structure, and it decreases the effectiveness of floating point filters. In Section 3.3 we propose infimaximal frames as a general technique for handling rays and lines. We propose to enclose the scene in a frame of infimaximal size and to clip rays and lines at the frame. Infimaximal frames support on-line algorithms, require no change in software structure, and cooperate well with floating point filters.

Our second example concerns map overlay. The texts [dBvKOS97, Section 2.3] and [MN99, Section 10.8] describe algorithms for maps with a single unbounded face, *i.e.*, all faces (except the unbounded face) are bounded by simple closed polygons. Again the algorithms readily generalize to subdivisions with more than one unbounded face, *e.g.*, Voronoi diagrams or arrangements of lines. Will implementations generalize? No, they do not. For example, the standard data structure for representing maps, namely doubly connected edge lists [PS85, dBvKOS97], assumes that all face

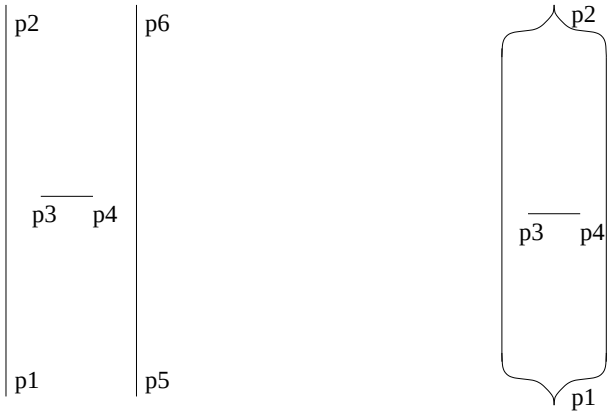



Figure 3.1: The left part shows a scene consisting of two parallel vertical segments and one horizontal segment. The sweep line encounters the endpoints in the order $p_1$, $p_2$, $p_3$, $p_4$, $p_5$, $p_6$. In the right part the vertical segments are extended to lines. In projective geometry, parallel lines share endpoints: Oriented projective geometry identifies $p_1$ with $p_5$ and $p_2$ with $p_6$ and standard projective geometry identifies all four points $p_1$, $p_2$, $p_5$, $p_6$. In either case, there is no order on the endpoints which would allow to sweep the scene.

cycles are closed and hence DCELs cannot even represent subdivisions with several unbounded faces in a direct way. Infimaximal frames offer a simple solution. Enclosing the scene in an infimaximal frame makes all faces (except the outside of the frame) finite and hence extends the use of DCELs to subdivisions with several unbounded faces.

This chapter is structured as follows. In Section 3.2 we discuss projective geometry and the inclusion in concrete geometric frames and argue that these approaches are insufficient. In Section 3.3 we introduce infimaximal frames and discuss their mathematics. In Section 3.4 we describe our implementation of infimaximal frames. In the three subsections we report on the program modules that realize geometry based on infimaximal frames. In Section 3.4.1 we work on the implementation of a polynomial arithmetic. In the following Section 3.4.2 we use the polynomial arithmetic together with the CGAL lower dimensional geometric kernel to implement a simple version of infimaximal frames. From this version we derive a filtered version in Section 3.4.3.

The end of Chapter Section 4 discusses our application experience. We use infimaximal frames in the implementation of Nef polyhedra and we compare the efficiency of our implementation of infimaximal frames with a realization of concrete geometric frames. We will see that there is no loss of efficiency and in some situations even a gain.

## 3.2 Alternative Approaches

We discuss projective geometry and the inclusion of the scene in a concrete geometric frame. We argue that projective geometry is unable to solve our problem and that the inclusion in a concrete geometric frame is unsatisfactory.

### Projective Geometry

Projective geometry provides points at infinity and hence, at first sight, seems to solve all our problems. There are two versions of projective geometry: the standard version [Cox87] and the oriented version of Stolfi [Sto91]. In the standard version, there is one point at infinity for every family of parallel lines, and in the oriented version, there are two points at infinity for every family of parallel lines. Neither version allows to sweep the configuration shown in the right part of Figure 3.1. In this configuration, a finite segment lies between two vertical parallel lines. Since the finite segment lies completely to the right of the left vertical line, the left vertical line should be swept before the finite segment. Similarly, the right vertical line should be swept after the segment. However, parallel lines share endpoints in projective geometry and hence there is no way to define a sweep order on the endpoints of lines and segments. We conclude that projective geometry is unable to solve our problem.

### Inclusion in a Concrete Geometric Frame

The following argument is typically used to show that an algorithm designed for segments can also handle rays and lines:

> Enclose the input scene in a large enough frame and clip rays and lines at the frame. Solve the problem for segments and translate back to rays and segments. The frame must be large enough such that no interesting geometry is lost. Adding and removing the frame are simple pre- and postprocessing steps which do not affect the asymptotic running time of the algorithm.

We next argue that inclusion in a concrete geometric frame is a bad implementation strategy.

- The frame must be large enough so that no interesting geometry is lost and hence the frame size can only be chosen once the input is completely known. Thus, on-line algorithms are excluded. Additionally, merging different scenes is non-trivial, if constructed with different frame sizes. It requires to change representations of points.
- Implementations have to be changed in a non-trivial way. We first need to make a pass over the data to determine an appropriate frame size. Next we clip rays and lines at the frame and replace them by segments. Then we run the algorithm for segments. Finally, we need to translate back.
- A large concrete frame size makes floating point filters ineffective. In the exact computation paradigm of computational geometry [OTU87, KLN91, Yap93, YD95, Sch99], all geometric predicates are evaluated exactly. Floating point filters are used to make exact computation efficient [FvW96, MN94, BFS98]. Floating point filters are most effective when point coordinates are small. Clipping rays and lines on a concrete frame introduces points with large coordinates which make filters less effective. Observe that in an arrangement of lines a single intersection with large coordinates will force the use of a large frame. Also observe, that lines with $k$ bit coefficients may intersect in points whose coordinates require $2k$ bits. Assume that our input consists of a set of lines. Two lines with equations $ax+by=f$ and $cx+dy=g$ intersect in point $((fd-gb)/(ad-bc),(ga-fc)/(ad-bc))$ (Cramer's rule).

It may seem that frame size can be changed dynamically. For example, one could define the frame size as a variable. Whenever a ray or line needs to be clipped, the current value of the variable is taken

as the frame size, and whenever interesting geometry happens outside the current frame, the value of the variable is increased. Also, when the frame size is increased, the coordinates of all points on the frame must be changed in order to maintain consistency and hence, the approach incurs a large overhead in time if the frame size needs to be adopted frequently. Infimaximal frames avoid this overhead.

## 3.3 Our approach

We use a frame of infimaximal size. More precisely, we enclose the scene in a square box with corners $NW(-R,R)$, $NE(R,R)$, $SE(R,-R)$, and $SW(-R,-R)$. We leave the value of $R$ unspecified and treat $R$ as an infimaximal number, *i.e.*, a number which is finite but larger than the value of any concrete real number. Infimaximal numbers are the counterpart of infinitesimal numbers as, for example, used in symbolic perturbation schemes [EM90].

Before we go into details, we argue that this proposal overcomes the deficiencies of the concrete frame approach.

- Since the value of $R$ is infimaximal, no interesting geometry lies outside the frame and on-line problems cause no difficulties. All scenes are constructed with the same infimaximal frame and hence merging scenes causes no problems.

- Implementations incur only minor changes. We will define new point classes and segment classes (extended points and extended segments, respectively). Extended points are either standard points or points on our infimaximal frame and extended segments are spanned by extended points. Thus extended segments can model standard segments, rays and lines. Many LEDA [LEDa] and CGAL [CGA] algorithms can operate on the new point and segment classes without much change, see Section 3.5 for examples.

- Filters stay effective up to larger input bit sizes. Point coordinates are polynomials in $R$ and the evaluation of geometric predicates amounts to determining the sign of the highest nonzero coefficient. These coefficients are smaller than the corresponding values arising in the computation with a fixed frame size, see Section 3.4 for details.

### 3.3.1 Frame Points and Extended Points

We define extended points in terms of their position with respect to the square box.

**Definition 1:** A *frame point* or *non-standard point* is a point on one of the four frame boundaries. A point on the left frame boundary has coordinates $(-R, f(R))$, where $f$ is a function with $|f(R)| \leq R$ for all sufficiently large $R$. Points on the other frame boundaries are defined analogously, *i.e.*, points on the right boundary have coordinates $(R, f(R))$, points on the lower boundary have coordinates $(f(R), -R)$, and points on the upper boundary have coordinates $(f(R), R)$. A *standard point* is simply a point in the affine plane and has coordinates $(x,y)$ with $x,y \in \mathbb{R}$. An *extended point* is either a standard point or a non-standard point.

Although the definition above makes sense for arbitrary function $f$, we restrict ourselves to linear functions in $R$, as this will suffice to model endpoints of rays and lines.

### 3.3.2 The Endpoints of Segments, Rays, and Lines

The endpoints of a segment are standard points, a ray has a standard endpoint and a non-standard endpoint and a line has two non-standard endpoints.

Consider a non-degenerate line $\ell$ with line equation $ax+by+c=0$ ($a^2+b^2\neq 0$). If $b=0$, the endpoints of the line are $(-c/a,\pm R)$. If $b\neq 0$, we have $y=mx+n$, where $m=-a/b$ and $n=-c/b$. If $|m|<1$, the line has endpoints $(\pm R,\pm mR+n)$, if $|m|>1$, the line has endpoints $(\pm R/m-n/m,\pm R)$, and if $|m|=1$, the line has endpoints $(-R,-mR+n)$ and $(R-nm,mR)$ if $sign(n)=sign(m)$, and has endpoints $(-R-nm,-mR)$ and $(R,mR+n)$ if $sign(n)\neq sign(m)$. The non-standard endpoint of a ray is determined similarly. We see that the coordinates of the endpoints of a line are simple linear functions in $R$, our infimaximal.

The endpoints of more complex geometric objects can be determined similarly, but the coordinate expressions become more complex. For example, the parabola $(y-x)^2=5x$ intersects the upper frame boundary in point $(R+5/2-\sqrt{5R+25/4},R)$ and the right frame boundary in point $(R,R-\sqrt{5R})$.

We use the coordinate approach described above also in our implementation, *i.e.*, the coordinates of frame points are linear functions in $R$ (recall that we are only dealing with lines and segments). We have also explored an alternative implementation strategy, namely to store a frame point as a reference to the underlying geometric object plus an indicator which selects the appropriate frame point. We found that this strategy leads to heavy case switching within geometric predicates as a point has four different representations (a standard point, a ray tip, the left endpoint of a line, and the right endpoint of a line) and hence a predicate operating on four points, *e.g.*, the *side_of_circle* predicate, would have to cope with up to $2^4$ cases.

The coordinate approach treats all cases uniformly. Moreover, it incurs no significant runtime penalty as we will see in Section 3.4.

### 3.3.3 Predicates on Extended Points

The flow of control in geometric algorithms is determined by the evaluation of geometric predicates. Important predicates are the lexicographic order of points (*compare_xy*), the orientation of a triple of points (*orientation*), and the incircle test for a quadruple of points (*side_of_circle*). Many[1] geometric predicates can be evaluated by computing the sign of a simple function defined on the coordinates of the points involved.

For example, the lexicographic order is simply a cascaded comparison of coordinates (sign of their difference), the orientation of three points is defined by

$$\text{orientation}(p1,p2,p3)=\text{sign}\begin{vmatrix}1&1&1\\x_1&x_2&x_3\\y_1&y_2&y_3\end{vmatrix}$$

and the side of circle predicate is given by

$$\text{side\_of\_circle}(p1,p2,p3,p4)=\text{sign}\begin{vmatrix}1&1&1&1\\x_1&x_2&x_3&x_4\\y_1&y_2&y_3&y_4\\x_1^2+y_1^2&x_2^2+y_2^2&x_3^2+y_3^2&x_4^2+y_4^2\end{vmatrix}$$

What happens when we apply these predicates to extended points? The value of the predicate will become the sign of a function in $R$. If this sign is independent of $R$ for all large values of $R$, the value

[1] The author knows of no predicate that is not.

of the predicate is well defined. For a large class of predicates and extended points this will be the case.

**Lemma 3.3.1:** If a geometric predicate is defined as the sign of a polynomial in point coordinates and the coordinates of extended points are polynomials in $R$, the value of the predicate when applied to the extended points is well-defined.

*Proof.* Assume that the predicate is defined as the sign of a polynomial $P$. Substituting the point coordinates into $P$ gives us a polynomial in $R$. For sufficiently large values of $R$, the sign of this polynomial is given by the sign of the highest nonzero coefficient. □

We give an example. Consider the orientation of two standard points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ and one non-standard point $p_3 = (-R, mR + n)$ on the left frame segment. We obtain:

$$\begin{aligned} \text{orientation}(p1, p2, p3) = \text{sign} \quad ( \quad & [m(x_2 - x_1) + (y_2 - y_1)]R \\ + \quad & [n(x_2 - x_1) + (x_1 y_2 - x_2 y_1)] \quad ) \end{aligned}$$

If the coefficient of $R$ is nonzero, its sign determines the orientation, and if the coefficient of $R$ is zero, the constant term determines the orientation. The latter is the case if the non-standard point is the endpoint of a line parallel to the line through $p_1$ and $p_2$.

**Lemma 3.3.2:** The values of the predicates *compare_xy*, *orientation*, and *side_of_circle* are well defined for endpoints of segments, rays and lines.

*Proof.* Predicates *compare_xy*, *orientation*, and *side_of_circle* are defined as signs of polynomials in point coordinates and the coordinates of endpoints of segments, rays, and lines are linear polynomials in $R$, our infimaximal. □

### 3.3.4 Extended Segments

We introduce extended segments as a unified view of segments, rays, and lines. Recall that our goal is to extend programs written for segments to rays and lines. Thus, we need a unified view of segments, rays, and lines.

A segment is defined by a pair of (standard) points. An extended segment is defined by a pair of extended points. We have to be a bit more careful. We cannot give meaning to every pair of extended points, but only to those pairs which correspond to a segment, ray, or line.

**Definition 2 (extended segment):** A pair of distinct extended points $(p_1, p_2)$ defines an *extended segment (esegment)* if one of the following conditions holds:

1. both points are *standard points*.
2. both points are *non-standard* and are the endpoints of a *common line*.
3. one point is *standard* and one is *non-standard* and they are the endpoints of a *common ray*.
4. both points are *non-standard points* and lie on the *same frame box segment*.

Extended segments defined by items 1) to 3) are called standard and extended segments defined by item 4) are called non-standard. Standard esegments correspond to objects of affine geometry, non-standard segments do not. For every fixed value of $R$, a non-standard segment corresponds to a well-defined geometric object.

### 3.3.5 Intersections of Extended Segments

An extended segment represent either a standard segment, a ray, a line, or a segment on one of the frame boundaries. We define the intersection point of two esegments as follows. If for every fixed sufficiently large value of $R$, the corresponding geometric objects intersect in a single point, this point is the point of intersection. Otherwise the intersection is undefined. Observe that if the intersection lies on the frame for every sufficiently large value of $R$, the intersection is indeed one of our frame points and hence this definition makes sense.

We next show that the standard analytical methods for handling intersections of segments apply to extended segments.

Consider two non-trivial segments $s_0 = (p_0, q_0)$ and $s_1 = (p_1, q_1)$ and their underlying lines $\ell_0$ and $\ell_1$. The segments intersect in a single point iff[2] the endpoints of $s_i$ do not lie on the same side[3] of $\ell_{1-i}$ for $i = 1, 2$. Thus, the test whether two segments intersect amounts to four evaluations of the orientation predicate. We have already argued that the orientation predicate extends and hence the test whether two segments intersect extends.

Consider next the computation of the intersection point $p$ of $s_0$ and $s_1$. The coordinates of $p$ are rational functions $r_x$ and $r_y$ of the coordinates of $p_0$ to $q_1$. Rational expressions $E_x$ and $E_y$ in the coordinates of $p_0$ to $q_1$ representing functions $r_0$ and $r_1$ are well known and easily obtained. Simply derive the line equations for $\ell_0$ and $\ell_1$ and then solve a linear system to obtain $E_x$ and $E_y$[4]:

$$\ell_i \equiv \quad a_i x + b_i y + c_i = 0$$

where

$$a_i = y_{pi} - y_{qi},\ b_i = x_{qi} - x_{pi},\ c_i = x_{pi} y_{qi} - x_{qi} y_{pi}$$

Then the point of intersection $p$ is defined by the expressions (Cramer's rule):

$$E_x = (b_0 c_1 - b_1 c_0)/(a_0 b_1 - a_1 b_0),\ E_y = (a_1 c_0 - a_0 c_1)/(a_0 b_1 - a_1 b_0)$$

What is the situation for two extended segments? The intersection point is an extended point and for every fixed value of $R$, $r_x(R)$ and $r_y(R)$ are the coordinates of the intersection point. If the intersection point is a standard point, $r_{x,y}(R)$ does not depend on $R$, and if the intersection point is non-standard[5], one of the functions $r_{x,y}(R)$ is the identity function and the other has absolute value at most $R$ for sufficiently large $R$. We may also apply the rational expressions $E_x$ and $E_y$ to the coordinates of the endpoints of the esegments $p_0$ to $q_1$. We obtain representations for rational functions in $R$, our infimaximal. For every fixed value of $R$, we have $r_{x,y}(R) = E_{x,y}(R)$ and hence the rational functions must simplify to the canonical representation of non-standard points. We have thus shown:

**Lemma 3.3.3:** Let *intersection*$(p1, p2, q1, q2)$ be the partial function that returns the coordinates of the intersection point of the segments $s(p_0, p_1)$ and $s(q_0, q_1)$. Then *intersection* is correct when applied to extended segments.

[2]For simplicity, we are ignoring the possibility that the underlying lines are identical and the two segments share an endpoint. The discussion is easily extended to also handle this situation.

[3]An oriented line has three sides: left, on, and right.

[4]Note that the rational expressions are not unique. One can easily expand the quotient by an arbitrary factor.

[5]Intersect a line or ray with the non-standard esegment that contains an endpoint.

## 3.4 Implementation

We implemented extended points in C++ and used them together with CGAL and LEDA. We report about the use in Section 3.5. Our implementation went through three versions.

In the first version an extended point was represented by a reference to the underlying geometric object plus an indicator which selects a frame point. Predicates and geometric constructions used case switching based on the representation. We soon realized that this approach is too cumbersome and that the complicated control structure of our predicates makes it difficult to ensure correctness.

In this section we report about the versions two and three. Both use the coordinate representation based on arithmetic in a polynomial ring. We implement a number type *RPolynomial* modeling $\mathbb{Z}[R]$ the Euclidean ring of polynomials in a variable $R$. Our type offers the ring operations $+$, $-$, $*$, polynomial division and the gcd operation, as described in [Coh93, Knu98], and a sign function. The sign function returns the sign of the highest nonzero coefficient.

We obtain extended points and segments by combining our number type *RPolynomial* with the geometry kernel of CGAL. The geometry kernels of CGAL are parameterized by an arithmetic type. Objects may either be represented by their Cartesian or their homogeneous coordinates. We use the homogeneous kernel as it only requires a ring type (the Cartesian kernel requires a number type that is a field). We instantiate two-dimensional homogeneous points with our number type *RPolynomial*, and add some additional construction code for points from standard points and oriented lines. No work is required for the geometric predicates as predicate evaluation amounts to sign computation of arithmetic expressions and we define the sign function of our ring type according to Section 3.3.3.

A slight modification is required for the intersection code. The CGAL kernel does not automatically cancel common factors in the representation of points, *i.e.*, it is not guaranteed that the gcd of the homogeneous coordinates of a point is equal to one[6]. For our situation this implies that point representations could contain redundant polynomial factors and hence non-linear polynomials. Correctness is not really impaired, but running times become miserable if one proceeds this way. We remedy the situation by insisting that representations are always in their reduced form, *i.e.*, whenever a point is constructed the gcd of the homogeneous coordinates is computed and a common factor is canceled. This ensures that the polynomials in point representations stay linear as argued in Section 3.3.5.

Our second implementation has the strong appeal of very modular programming and thereby its correctness was very simple to achieve. We have used it heavily as a backup checker for the more elaborate techniques used in our third version.

In the third version we optimize the representation of points and the evaluation of predicates. This forces us to write our own classes *epoint* and *esegment* and to write code for the evaluation of predicates. In the representation of points we exploit that the normalizing coordinate is an integer (and never a polynomial of degree one). In order to optimize the evaluation of predicates, we derive closed form expressions for the polynomials arising in the predicates and incorporate filter technology.

### 3.4.1 Polynomials in one variable

We present the implementation of a simple polynomial type *RPolynomial* in one variable. The interface is specified in the manual page on page 190. Let *NT* be a either a field number type or an Euclidean ring number type[7]. We use $NT[x]$ to represent the polynomial ring in one variable. For a polynomial $p = \sum_{i=0}^{d} a_i x^i \in NT[x]$ we store its coefficients along its rising exponents $coeff[i] = a_i$ in

[6] It cannot do so since the notion of gcd does not make sense for every ring type.

[7] An Euclidean ring type is a ring that additionally offers division with remainder and as such is a unique factorization domain.

a vector of size $d+1$. We keep the invariant that $a_d \neq 0$ and do not allow modifying access to the coefficients via the interface. Flexibility in the creation of polynomials is achieved via iterator ranges which can specify a sequence of coefficients of a polynomial. We offer basic arithmetic operations like $+, -, *$, as well as destructive self modifying operations $+=, -=, *=$. When working destructively we need a cloning scheme to cope with the alias effects of one common representation referenced by several handles. For number types that are fields or Euclidean rings we also offer polynomial division. For the field types this can be done directly by so called Euclidean division. For the number types that are Euclidean rings we provide it via so called pseudo division. Based on that operation we also provide a gcd-operation on the ring $NT[x]$.

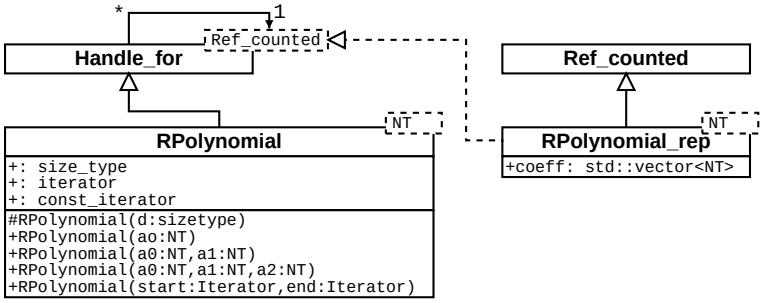



Figure 3.2: Some details of the handle scheme. *RPolynomial_rep*<*NT*> is derived from *Ref_counted* and thereby a model of the template parameter $T$ of *Handle_for*<*T*>. It stores the coeffi cients in an STL vector *coeff*. *RPolynomial*<*NT*> is derived from *Handle_for*< *RPolynomial_rep*<*NT*> >. A *Ref_counted* object carries the reference variable, *Handle_for*<*T*> provides the copy construction and assignment mechanisms.

### Implementation

Polynomials are implemented by using a smart-pointer scheme. First we implement the common representation class storing an *NT* vector. The whole smart-pointer scheme is shown in Figure 3.2. For the representation class we keep the invariant that the coefficient vector *coeff* is always reduced such that the highest-order entry is nonzero except when the degree is zero. In this case we allow a zero entry. By doing so the degree of the polynomial is always *coeff.size*( ) − 1. To keep our invariant we *reduce* the coefficient representation after each construction and arithmetic modification, which basically means we shrink the coefficient vector until the last entry is nonzero or until the polynomial is constant. In this description we only show the most interesting facets of the implementation. The whole implementation project is presented in our implementation report. The *pop_back*( ) operation of the STL vector nicely supports the *reduce*( ) method.

⟨*rep interface*⟩+≡
```
void reduce()
{ while ( coeff.size()>1 && coeff.back()==NT(0) ) coeff.pop_back(); }
```

In this chunk we present the method interface of *RPolynomial*<>. The array operators offer read-only access. For manipulation we offer a protected method *coeff*(*int*), that is only for internal use. Evaluation, sign, and content are simple to realize.

⟨*public interface*⟩≡
```
const_iterator begin() const { return ptr->coeff.begin(); }
const_iterator end()   const { return ptr->coeff.end(); }

int degree() const
{ return ptr->coeff.size()-1; }

const NT& operator[](unsigned int i) const
{ CGAL_assertion( i<(ptr->coeff.size()) );
  return ptr->coeff[i]; }

const NT& operator[](unsigned int i)
{ CGAL_assertion( i<(ptr->coeff.size()) );
  return ptr->coeff[i]; }

NT eval_at(const NT& r) const
{ CGAL_assertion( degree()>=0 );
  NT res = ptr->coeff[0], x = r;
  for(int i=1; i<=degree(); ++i)
  { res += ptr->coeff[i]*x; x*=r; }
  return res;
}

CGAL::Sign sign() const
{ const NT& leading_coeff = ptr->coeff.back();
  if (leading_coeff < NT(0)) return (CGAL::NEGATIVE);
  if (leading_coeff > NT(0)) return (CGAL::POSITIVE);
  return CGAL::ZERO;
}

bool is_zero() const
{ return degree()==0 && ptr->coeff[0]==NT(0); }

RPolynomial<NT> abs() const
{ if ( sign()==CGAL::NEGATIVE ) return -*this; return *this; }

NT content() const
{ CGAL_assertion( degree()>=0 );
  return gcd_of_range(ptr->coeff.begin(),ptr->coeff.end());
}
```

### Arithmetic Ring Operations

Next we come to the implementation of basic arithmetic operations. The negation is trivial. We just iterate over the coefficient array and invert each sign. Addition $p1 + p2$ is also easy. Just add all coefficients of the two monomials with the same degree. Note however that the polynomials themselves might have different degree, such that we have to copy all coefficients in the range $min(d_{p1}, d_{p2}) + 1$ up to $max(d_{p1}, d_{p2})$ into the result. Afterwards we have to reduce the coefficient vector. The subtraction routine is symmetric. We only have to deal with the different sign of *p2*.

⟨*polynomial implementation*⟩+≡
```
template <class NT>
RPolynomial<NT> operator + (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
{
  typedef typename RPolynomial<NT>::size_type size_type;
```

```
  CGAL_assertion(p1.degree()>=0 && p2.degree()>=0);
  bool p1d_smaller_p2d = p1.degree() < p2.degree();
  int min,max,i;
  if (p1d_smaller_p2d) { min = p1.degree(); max = p2.degree(); }
  else                 { max = p1.degree(); min = p2.degree(); }
  RPolynomial<NT>  p( size_type(max + 1));
  for (i = 0; i <= min; ++i ) p.coeff(i) = p1[i]+p2[i];
  if (p1d_smaller_p2d)  for (; i <= max; ++i ) p.coeff(i)=p2[i];
  else /* p1d >= p2d */ for (; i <= max; ++i ) p.coeff(i)=p1[i];
  p.reduce();
  return p;
}
```

Multiplication is also straightforward. The degree formula tells us that $p1 * p2$ has degree $p1.degree( ) + p2.degree( )$. We just allocate a polynomial $p$ of corresponding size initialized to zero and add the products of all monomials $p1[i] * p2[j]$ for $0 \le i \le d_{p1}$, $0 \le j \le d_{p2}$ slotwise to $a_{i+j}$.

⟨*polynomial implementation*⟩+≡

```
template <class NT>
RPolynomial<NT> operator * (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
{
  typedef typename RPolynomial<NT>::size_type size_type;
  CGAL_assertion(p1.degree()>=0 && p2.degree()>=0);
  RPolynomial<NT>  p( size_type(p1.degree()+p2.degree()+1) );
  // initialized with zeros
  for (int i=0; i <= p1.degree(); ++i)
    for (int j=0; j <= p2.degree(); ++j)
      p.coeff(i+j) += (p1[i]*p2[j]);
  p.reduce();
  return p;
}
```

**Polynomial Division and Reduction**

Next we implement polynomial division operations. See also the books of Cohen [Coh93] or Knuth [Knu98] for a thourough treatment. The result of our division operation $p_1/p_2$ in $NT[x]$ is defined as the polynomial $p_3$ such that $p_1 = p_2 p_3$. In case there is no such polynomial the result is undefined.

The implementation of *operator/* depends on the number type plugged into the template. To provide the division we implement two division operations *pseudo_div* and *euclidean_div*. The first operation works with ring number types providing a *gcd* operation, so called *unique factorization domains*. The second operation works with polynomials over *field* number types. To separate our number types we introduce a traits class providing tags to choose one or the other code variant. In the header file of *RPolynomial* there are three predefined class types *ring_or_field_dont_know*, *ring_with_gcd*, and *field_with_div*. As a prerequisite a user has just to specialize the class template *ring_or_field*<> to the number type that she wants to plug into *RPolynomial*<>. For the LEDA integer type this can be done as follows

```
template <>
struct ring_or_field<leda_integer> {
  typedef ring_with_gcd kind;
  static leda_integer gcd(const leda_integer& a, const leda_integer& b)
  { return ::gcd(a,b); }
};
```

In case of a Euclidean ring the class *ring_or_field<RT>* has to provide the gcd operation of two *RT* operands as a static method [8]. Based on this number type flag *RPolynomial<leda_integer>* provides the division operation with the help of *pseudo_div* based on the *gcd* operation of *leda_integer*. For users not providing the *ring_or_field<>* specialization an error message is raised.

The division operator is implemented depending on the number type *NT*. Our number type traits *ring_or_field<NT>* provides a tag type to specialize it via three overloaded methods *divop*( ) that are implemented below.

⟨*polynomial implementation*⟩+≡

```
template <class NT> inline
RPolynomial<NT> operator / (const RPolynomial<NT>& p1,
                            const RPolynomial<NT>& p2)
{ return divop(p1,p2,ring_or_field<NT>::kind()); }
```

### Field Number Types

We first implement standard polynomial division. Starting from polynomials $f$ and $g$ we determine two polynomials $q$ and $r$ such that $f = q * g + r$ where $d_r \leq d_g$.

⟨*polynomial statics*⟩≡

```
template <class NT>
void RPolynomial<NT>::euclidean_div(
  const RPolynomial<NT>& f, const RPolynomial<NT>& g,
  RPolynomial<NT>& q, RPolynomial<NT>& r)
{
  r = f; r.copy_on_write();
  int rd=r.degree(), gd=g.degree(), qd(0);
  if ( rd < gd ) { q = RPolynomial<NT>(NT(0)); }
  else { qd = rd-gd+1; q = RPolynomial<NT>(size_t(qd)); }
  while ( rd >= gd ) {
    NT S = r[rd] / g[gd];
    qd = rd-gd;
    q.coeff(qd) += S;
    r.minus_offsetmult(g,S,qd);
    rd = r.degree();
  }
  CGAL_postcondition( f==q*g+r );
}
```

We need an operation which allows us to subtract a polynomial $s$ which is the product of a polynomial $p = \sum_{i=0}^{d} a_i x^i$ and a monomial $m = bx^k$. The result is $mp = \sum_{i=0}^{d+k} b\tilde{a}_i x^i$ where $\tilde{a}_i = 0$ for $0 \leq i < k$

[8] This makes life easier when working with compilers that lack Koenig-lookup.

and $\tilde{a}_{i+k} = a_i$ for $0 \le i \le d$. We implement this by shifting the coefficients of $p$ by $k$ places while multiplying them by $b$ and leave the lower $k$ entries of the resulting polynomial zero.

⟨*offset multiplication*⟩≡
```
void minus_offsetmult(const RPolynomial<NT>& p, const NT& b, int k)
{ CGAL_assertion(!ptr->is_shared());
  RPolynomial<NT> s(size_type(p.degree()+k+1)); // zero entries
  for (int i=k; i <= s.degree(); ++i) s.coeff(i) = b*p[i-k];
  operator-=(s);
}
```

Now we can just specialize *divop* in its third argument:

⟨*polynomial implementation*⟩+≡
```
template <class NT>
RPolynomial<NT> divop (const RPolynomial<NT>& p1,
                       const RPolynomial<NT>& p2,
                       field_with_div)
{ CGAL_assertion(!p2.is_zero());
  if (p1.is_zero()) return 0;
  RPolynomial<NT> q,r;
  RPolynomial<NT>::euclidean_div(p1,p2,q,r);
  CGAL_postcondition( (p2*q+r==p1) );
  return q;
}
```

### Unique factorization domains

Polynomial division avoiding using the notion of an inverse that is present in fields, is not so trivial. We first introduce the algebraic notions necessary for some theoretic results. We start with the introduction of divisibility and greatest common divisors. Let $f = \sum_{i=0}^{d_f} a_i x^i$, $g = \sum_{i=0}^{d_g} b_i x^i$ be polynomials of *degree* $d_f$ and $d_g$. We assume the *leading coefficient* $a_{d_f}$ and $b_{d_g}$ to be nonzero and thereby degree defining.

**Definition 3:** A *commutative ring* $\mathcal{R}$ with unit 1 and containing no zero divisors is called an *integrity domain*. An integrity domain where every nonzero element is either a *unit* or has a unique representation[9] as a product of primes is called a *unique factorization domain*.

The integers $\mathbb{Z}$ are our default example of a unique factorization domain. One example for a ring that is no unique factorization domain is $\mathbb{Z}/4\mathbb{Z}$ which contains the zero divisor 2.

**Definition 4:** Let $\mathcal{R}$ be an integrity domain. $\mathcal{K} = Quot(\mathcal{R})$, $\mathcal{K}^* = \mathcal{K} - \{0\}$. Let $a, b \in \mathcal{K}^*$.

$$(1) \quad a|_{\mathcal{R}} b \text{ "}a \text{ divides } b \text{ in } \mathcal{R}\text{"} \quad :\Leftrightarrow \exists c \in R : b = ac$$
$$(2) \quad a \sim_{\mathcal{R}} b \;:\Leftrightarrow\; a|_{\mathcal{R}} b \wedge b|_{\mathcal{R}} a$$

Let $d \in \mathcal{K}^*$. $d$ is called $\gcd_{\mathcal{K}}(a,b) \;:\Leftrightarrow$

$$(1) \quad d|_{\mathcal{R}} a \wedge d|_{\mathcal{R}} b$$
$$(2) \quad \forall c \in \mathcal{K}^* : c|_{\mathcal{R}} a \wedge c|_{\mathcal{R}} b \Longrightarrow c|_{\mathcal{R}} d$$

[9] Uniqueness up to permutations and multiplication with units.

$d$ is determined uniquely except for multiplication with units.

The following lemma assures that we can divide $f$ by $g$ in the integral domain when we accept an expansion of $f$ by a power of the leading coefficient of $g$.

**Lemma 3.4.1:** Let $\mathcal{R}$ be a ring and $f,g \in \mathcal{R}[x], g \neq 0$. Let $b$ be the leading coefficient of $g$. Then there are polynomials $q,r \in \mathcal{R}[x]$, such that

$$b^s f = qg + r$$

where either $r = 0$ or $r \neq 0$ and $d_r < d_g$ and the integer $s = 0$ if $f = 0$ and $s = \max\{0, d_f - d_g + 1\}$ if $f \neq 0$. If $b$ is no zero divisor in $\mathcal{R}$ then $q$ and $r$ are uniquely defined.

*Proof.* **Existence** — In case $f = 0$ and $f \neq 0, d_f < d_g$ the pair $q = 0, r = f$ is a trivial solution. Let $f \neq 0$, $\Delta(f,g) := d_f - d_g \geq 0$. We show the existence of $q,r$ by induction on $\Delta(f,g)$. Let $q,r$ exist for polynomials $f,g, f \neq 0, \Delta(f,g) < \Delta_0, \Delta_0 \geq 0$. Now let $f \in \mathcal{R}[x]$, $f \neq 0, \Delta(f,g) = \Delta_0$ and $a$ be the leading coefficient of $f$. We look at $f' := bf - ax^{\Delta_0} g$. Either $f' = 0$ or $f' \neq 0$ but $d_{f'} < d_f, \Delta(f',g) < \Delta_0$. By the induction hypothesis there are $r', q' \in \mathcal{R}[x]$ such that $b^{s'} f = q'g + r'$ where either $r' = 0$ or $r \neq 0$ and $d_{r'} < d_g$ and the non-negative integer $s' \leq d_f - d_g$. Thus $b^{s'+1} f = (b^{s'} ax^{\Delta_0} + q')g + r'$. Because of $s' + 1 \leq d_f - d_g + 1$ the existence of $q$ and $r$ is clear.

**Uniqueness** — Let $b$ be no zero divisor in $\mathcal{R}$ and assume that there is another representation $b^s f = \tilde{q}g + \tilde{r}$ where either $\tilde{r} = 0$ or $\tilde{r} \neq 0$ and $d_q < d_{\tilde{q}}$ By subtracting one from the other we obtain $(q - \tilde{q})g = \tilde{r} - r$. As the $b$ is no zero divisor in $\mathcal{R}$ the degree formula tells us that $\deg(q - \tilde{q})g = \deg(q - \tilde{q}) + d_g \geq d_g$. Thus $\tilde{r} - r \neq 0$ and $\deg(\tilde{r} - r) \geq d_g$. On the other hand $\tilde{r} - r$ is the difference of two polynomials of degree smaller than $d_g$ and therefore $\deg(\tilde{r} - r) < d_g$. From the contradiction we obtain $q - \tilde{q} = \tilde{r} - r = 0$. □

See any algebra book like [RSV84] for more details. Knuth [Knu98] calls the algorithm based on the above lemma *pseudo-division*. According to this lemma one can determine $q$ and $r$ within the ring without resorting to the quotient field. We follow the construction in the proof in reverse order to reduce $f$ down to $r$. We use *minus_offsetmult* for the reduction from $f$ to $f'$.

⟨*polynomial statics*⟩+≡

```
template <class NT>
void RPolynomial<NT>::pseudo_div(
  const RPolynomial<NT>& f, const RPolynomial<NT>& g,
  RPolynomial<NT>& q, RPolynomial<NT>& r, NT& D)
{
  int fd=f.degree(), gd=g.degree();
  if ( fd<gd )
  { q = RPolynomial<NT>(0); r = f; D = 1;
    CGAL_postcondition(RPolynomial<NT>(D)*f==q*g+r); return;
  }
  // now we know fd >= gd and f>=g
  int qd=fd-gd, delta=qd+1, rd=fd;
  q = RPolynomial<NT>( size_t(delta) );
  NT G = g[gd]; // highest order coeff of g
  D = G; while (--delta) D*=G; // D = G^delta
  RPolynomial<NT> res = RPolynomial<NT>(D)*f;
  while (qd >= 0) {
    NT F = res[rd]; // highest order coeff of res
```

```
    NT t = F/G;     // ensured to be integer by multiplication of D
    q.coeff(qd) = t;    // store q coeff
    res.minus_offsetmult(g,t,qd);
    if (res.is_zero()) break;
    rd = res.degree();
    qd = rd - gd;
  }
  r = res;
  CGAL_postcondition(RPolynomial<NT>(D)*f==q*g+r);
}
```

Finally we specialize *divop* for unique factorization domains.

⟨*polynomial implementation*⟩+≡

```
template <class NT>
RPolynomial<NT> divop (const RPolynomial<NT>& p1, const RPolynomial<NT>& p2,
                       ring_with_gcd)
{ CGAL_assertion(!p2.is_zero());
  if (p1.is_zero()) return 0;
  RPolynomial<NT> q,r; NT D;
  RPolynomial<NT>::pseudo_div(p1,p2,q,r,D);
  CGAL_postcondition( (p2*q+r==p1*RPolynomial<NT>(D)) );
  return q/=D;
}
```

For the reduction of polynomials we finally implement the greatest common divisor method as introduced by Euclid. For two elements $a, b \in \mathcal{R}$ Euclids algorithm uses the reduction $\gcd(a,b) \sim_{\mathcal{R}} \gcd(b, a \mod b)$.

**Definition 5:** For a polynomial $f = \sum_{i=0}^{d} a_i x^i$ we define its *content* as $\mathrm{cont}(f) = \gcd(a_0, \ldots, a_d)$ and its *primitive part* as $\mathrm{pp}(f) = f/\mathrm{cont}(f)$.

Again the content of a polynomial is only unique up to multiplication by units of $\mathcal{R}$. Note that $\mathrm{cont}(f)$ is a divisor of all coefficients of $f$ in $\mathcal{R}$ and therefore the division is reducing the representation of $f$ such that $\mathrm{cont}(\mathrm{pp}(f)) = 1$. The following lemma tells us something about the composition of the gcd of two polynomials from the contents and the primitive parts. A polynomial whose content is 1 is called *primitive*.

**Lemma 3.4.2 (Gauss):** The product of two *primitive* polynomials $f$ and $g$ over a unique factorization domain is again *primitive*. Moreover let $f$ and $g$ be two general polynomials over a unique factorization domain $\mathcal{R}$. Then $\mathrm{cont}(\gcd(f,g)) \sim_{\mathcal{R}} \gcd(\mathrm{cont}(f), \mathrm{cont}(g))$ and $\mathrm{pp}(\gcd(f,g)) \sim_{\mathcal{R}} \gcd(\mathrm{pp}(f), \mathrm{pp}(g))$.

*Proof.* Let $f = \sum_{i=0}^{d_f} a_i x^i, g = \sum_{i=0}^{d_g} b_i x^i$ be primitive polynomials. We show for any prime $p$ of the domain that it does not divide all the coefficients of $f \cdot g$. For both polynomials we chose the smallest indices $j$ and $k$ for which $p$ does divide $(a_i)_i$ and $(b_i)_i$. We then examine the coefficient of $x^{j+k}$ of $f \cdot g$:

$$a_j b_k + a_{j+1} b_{k-1} + \cdots + a_{j+k} b_0 + a_{j-1} b_{k+1} + \cdots + a_0 b_{k+j}$$

As $p$ divides only the first term and none of the following terms, $p$ does not divide the sum.

From the above we can deduce for general polynomials $f$ and $g$ that $\mathrm{pp}(fg) \sim_{\mathcal{R}} \mathrm{pp}(f)\mathrm{pp}(g)$. The product $fg$ can be decomposed as $fg = \mathrm{cont}(f)\mathrm{pp}(f)\mathrm{cont}(g)\mathrm{pp}(g) = \mathrm{cont}(f)\mathrm{cont}(g)\mathrm{pp}(g)\mathrm{pp}(f)$ and $fg = \mathrm{cont}(fg)\mathrm{pp}(fg)$. Thereby we can deduce that $\mathrm{cont}(fg) \sim_{\mathcal{R}} \mathrm{cont}(f)\mathrm{cont}(g)$.

Now assume that $h \sim_{\mathcal{R}} \gcd(f,g)$ and thus $f = hF$ and $g = hG$ for some polynomials $F$ and $G$ from $\mathcal{R}[x]$. By the previous result we get $\mathrm{cont}(f) \sim_{\mathcal{R}} \mathrm{cont}(h)\mathrm{cont}(F)$ and $\mathrm{cont}(g) \sim_{\mathcal{R}} \mathrm{cont}(h)\mathrm{cont}(G)$ and thereby $\mathrm{cont}(\gcd(f,g)) \sim_{\mathcal{R}} \mathrm{cont}(h) \sim_{\mathcal{R}} \gcd_{\mathcal{R}}(\mathrm{cont}(f), \mathrm{cont}(g))$. The latter equality follows from the fact that $\gcd_{\mathcal{R}}(\mathrm{cont}(F), \mathrm{cont}(G)) \sim_{\mathcal{R}} 1$ due to the properties of $h$ in the decomposition of $f$ and $g$. A similar argument shows that $\mathrm{pp}(\gcd(f,g)) \sim_{\mathcal{R}} \gcd(\mathrm{pp}(f), \mathrm{pp}(g))$. □

This result simplifies the problem and allows us to keep the size of the coefficients of the polynomials as small as possible. An elaborate treatment of the topic can be found in [Coh93, Knu98].

By the above lemma we obtain the following strategy. First calculate $F = \gcd_{\mathcal{R}}(\mathrm{cont}(f), \mathrm{cont}(g))$ by the gcd routine on the ring number type *NT*. Reduce both polynomials by their content to their primitive parts $f^\circ = \mathrm{pp}(f)$ and $g^\circ = \mathrm{pp}(g)$.

Then reduce $\gcd(f^\circ, g^\circ) \sim_{\mathcal{R}} \gcd(g^\circ, f^\circ \mod g^\circ)$. However our pseudo-division *pseudo_div* only allows reductions of the form $(Df^\circ, g^\circ)$ to $(g^\circ, Df^\circ \mod g^\circ)$ where $D = b^s$ as described above. This is ok though as $\gcd(Df^\circ, g^\circ) \sim_{\mathcal{R}} \gcd_{\mathcal{R}}(\mathrm{cont}(Df^\circ), \mathrm{cont}(g^\circ))\gcd(\mathrm{pp}(Df^\circ), \mathrm{pp}(g^\circ)) \sim_{\mathcal{R}} \gcd_{\mathcal{R}}(D,1)\gcd(f^\circ, g^\circ) \sim_{\mathcal{R}} \gcd(f^\circ, g^\circ)$. The final result of the Euclidean reduction delivers $d^\circ = \gcd(f^\circ, g^\circ)$. We obtain the desired result $\gcd(f,g) = F \cdot d^\circ$.

⟨*polynomial statics*⟩+≡

```
template <class NT>
RPolynomial<NT> RPolynomial<NT>::gcd(
  const RPolynomial<NT>& p1, const RPolynomial<NT>& p2)
{
  if ( p1.is_zero() )
    if ( p2.is_zero() ) return RPolynomial<NT>(NT(1));
    else return p2.abs();
  if ( p2.is_zero() )
    return p1.abs();

  RPolynomial<NT> f1 = p1.abs();
  RPolynomial<NT> f2 = p2.abs();
  NT f1c = f1.content(), f2c = f2.content();
  f1 /= f1c; f2 /= f2c;
  NT F = ring_or_field<NT>::gcd(f1c,f2c);
  RPolynomial<NT> q,r; NT M=1,D;
  bool first = true;
  while ( ! f2.is_zero() ) {
    RPolynomial<NT>::pseudo_div(f1,f2,q,r,D);
    if (!first) M*=D;

    r /= r.content();
    f1=f2; f2=r;
    first=false;
  }

  return RPolynomial<NT>(F)*f1.abs();
}
```

Finally we provide a gcd calculation routine for a sequence of numbers. This routine requires the existence of a *gcd* operation as provided by our number type traits *ring_or_field*<*NT*>.

⟨*gcd of range*⟩≡

```
template <class Forward_iterator>
typename std::iterator_traits<Forward_iterator>::value_type
gcd_of_range(Forward_iterator its, Forward_iterator ite)
{ CGAL_assertion(its!=ite);
  typedef typename std::iterator_traits<Forward_iterator>::value_type NT;
  NT res = *its++;
  for(; its!=ite; ++its) res =
    (*its==0 ? res : ring_or_field<NT>::gcd(res, *its));
  if (res==0) res = 1;
  return res;
}
```

In this presentation we omit the details of input and output operations and all additional technical requirement necessary to make *RPolynomial*<> a CGAL number type. The type is specialized for the built-in number types *int* and *double*. This is necessary as the method interface for the general template class has to have methods for *NT* as well as for *int* and *double* to resolve construction ambiguities when using numeric constants of the built-in types. On the other hand these initialization methods collide when the general template would be instantiated for these types. Therefore the specialized class types have an interface avoiding the collisions.

### 3.4.2 Simple implementation - Standard kernel plus polynomial number type

We implement planar extended points by a homogeneous component representation in a polynomial ring type which provides standard ring operations like $+, -, *$. The definition of extended points puts constraints on the kind of polynomials representing the coordinates. We have seen that our extensible predicates are defined via polynomials in the coordinate polynomials and as such are extensible via the limit process on polynomials. Going to infinity the value of a polynomial is determined by the highest-order nonzero coefficient.

Using extensible predicates on an input set of extended points in the execution of an algorithm we can determine a concrete value $R_0$ which ensures their extendibility for all $R \geq R_0$ (for each evaluation determine one $R_i$ and take the maximum of all). Plugging $R_0$ into all coordinate polynomials leads back to standard affine geometry and standard predicates. This gives us the possibility to argue also about the correctness of our algorithms. If the algorithm is proven to be correct for standard geometry and it computes a certain output then it will also calculate some extended geometric result when plugging in extended points and when all geometric predicates are extensible.

Note that in this way we can design algorithms that use ray like structures much simpler by enclosing finite structures into the box $F$ and pruning the rays by means of the frame in a ray tip. The calculation with the extended points makes algorithmic decisions trivial if the predicates we use are extensible in the above sense.

In this section we will describe how extended points are stored: they are composed from the 2D CGAL kernel point type *Homogeneous*<...>::*Point_2* and the polynomial ring number type *RPolynomial*<...>. We also describe how the affine world of standard points and rays interacts with the unifying concept extended point. This interaction has two directions: the construction of an extended point from a standard object (point or ray) and the reversal extraction depending on the character of the extended point. Afterwards, we show how simple it is to implement predicates and the intersection construction on top of the genericity of CGAL's standard kernel. We will encounter the problem of

simplification of polynomials there. And finally, we give some details about visualization issues of extended objects.

We often use the short term *epoint* to denote extended points. Each epoint is either a standard point, one of the corner ray points or lies in the relative interior of one of the frame segments.

**Extended Points**

The tip of a ray $l$ can be described in two ways. First in form of its underlying oriented line equation $ax+by+c=0$. But also by its point-vector form $p=p_0+\lambda d$. The former is the standard representation of lines. The latter is more suitable to explore the character of the corresponding extended point.

Starting from the second representation we have two points $p_0$ and $p_0+d$ on the line. Now all points on the line can also be described by the determinant equation:

$$\begin{vmatrix} 1 & 1 & 1 \\ x_0 & x_0+d_x & x \\ y_0 & y_0+d_y & y \end{vmatrix} = 0$$

and developing this by the last column leads us to $-d_y x+d_x y+(x_0 d_y-y_0 d_x)=0$. Thus the direction vector is:

$$d=\begin{pmatrix} d_x \\ d_y \end{pmatrix}=\begin{pmatrix} b \\ -a \end{pmatrix}$$

A point on the line specified by the line equation is:

$$p_0=\begin{cases} (0,-c/b) & b\neq 0 \\ (-c/a,0) & a\neq 0 \end{cases}$$

Both $a$ and $b$ cannot be zero. In the following assume *Line_2* to be a model for the CGAL standard geometric kernel.

⟨*line conversion methods*⟩≡
```
static RT dx(const Line_2& l) { return l.b(); }
static RT dy(const Line_2& l) { return -l.a(); }
```

Depending on the slope $m=d_y/d_x (d_x\neq 0)$ of a line $l$ we can define its vertical distance to the origin. If $d_x\neq 0$ ($|m|\neq\infty$) then the ordinate intersection $d_o$ determines that distance $d_o=-c/b$.

⟨*line conversion methods*⟩+≡
```
static FT ordinate_distance(const Line_2& l)
{ return Kernel::make_FT(-l.c(),l.b()); }
```

We introduce enumeration specifiers that describe extended points.

⟨*enumerate extended point character*⟩≡
```
enum Point_type { SWCORNER=1, LEFTFRAME, NWCORNER,
                  BOTTOMFRAME, STANDARD, TOPFRAME,
                  SECORNER, RIGHTFRAME, NECORNER };
```

Now if we look at a non-standard point $p$ with underlying line equation $ax+by+c=0$ the frame segment which is hit by the ray tip is determined by the slope and in case $|m|=1$ by the distance $d_o$ defined above. Look for example at a non-standard point hitting the left frame segment. This is generally the case if $d_x<0$ and $|m|<1$. The latter is equivalent to the condition $|d_x|>|d_y|$. A special case is $|m|=1$. Then, we only hit the left segment if either $m=-1\wedge d_o<0$ or $m=1\wedge d_o>0$. The latter can be checked by $sign(d_y)==-sign(d_o)$. Note that because $|m|\leq 1$ the line indeed intersects the $y$-axis. The other cases follow by symmetric reasoning.

⟨*line conversion methods*⟩+≡

```
static Point_type determine_type(const Line_2& l)
{
  RT adx = abs(dx(l)), ady = abs(dy(l));
  int sdx = sign(dx(l)), sdy = sign(dy(l));
  int cmp_dx_dy = compare(adx,ady), s(1);
  if (sdx < 0 && ( cmp_dx_dy > 0 || cmp_dx_dy == 0 &&
      sdy != (s = sign(ordinate_distance(l))))) {
    if (0 == s) return ( sdy < 0 ? SWCORNER : NWCORNER );
    else        return LEFTFRAME;
  } else if (sdx > 0 && ( cmp_dx_dy > 0 || cmp_dx_dy == 0 &&
             sdy != (s = sign(ordinate_distance(l))))) {
    if (0 == s) return ( sdy < 0 ? SECORNER : NECORNER );
    else        return RIGHTFRAME;
  } else if (sdy < 0 && ( cmp_dx_dy < 0 || cmp_dx_dy == 0 &&
             ordinate_distance(l) < FT(0))) {
    return BOTTOMFRAME;
  } else if (sdy > 0 && ( cmp_dx_dy < 0 || cmp_dx_dy == 0 &&
             ordinate_distance(l) > FT(0))) {
    return TOPFRAME;
  }
  CGAL_assertion_msg(false," determine_type: degenerate line.");
  return (Point_type)-1; // never come here
}
```

All the operations above are packaged into the class *Line_to_epoint<R>*, where $R$ is a model of the CGAL standard 2d geometric kernel. From $R$ we derive the types *RT*, *FT*, and *Line_2* as used in the code.

Any non-standard point can be expressed as a pair of two polynomials in a variable $R$ — our infimaximal symbolic number. Let's look at our example again. Our point $p$ on the left frame segment supported by the line $ax+by+c=0$ can be described by the tuple $(-R, a/bR-c/b)$. Accordingly, a ray tip on the upper frame segment can be described by $(-b/aR-c/a, R)$. Note that the denominators are nonzero in both cases due to their frame position. Thus we can store epoints in terms of linear polynomials $mR+n$. For standard points the polynomials are just constant with $m=0$. We give the representation of all points in homogeneous representation, such that all coefficients can be

represented by a ring type.

$$\begin{array}{ll} \text{STANDARD} & p = (x,y,w) \\ \text{CORNER} & p = (\pm R, \pm R, 1) \\ \text{LEFTFRAME} & p = (-bR, aR - c, b) \\ \text{RIGHTFRAME} & p = (bR, -aR - c, b) \\ \text{BOTTOMFRAME} & p = (bR - c, -aR, a) \\ \text{TOPFRAME} & p = (-bR - c, aR, a) \end{array} \tag{3.1}$$

The general representation can be taken to be $p = (m_x R + n_x, m_y R + n_y, w)$ where $m_{x,y}, n_{x,y}, w$ are objects of a ring number type. We provide the functionality of extended points bundled into an extended geometry kernel. This kernel carries the types, predicates, and constructions that we need in our algorithms. The kernel concept is specified in the manual page *ExtendedKernelTraits_2* of the appendix.

**A decorator wraps functionality**

We obtain the extended point class by plugging our polynomial arithmetic type into the standard homogeneous point type from the CGAL kernel. We create a traits class *Extended_homogeneous<RT>* that carries all types and methods that are used in our algorithmic framework.

To ensure the special character of homogeneous points concerning their coordinates and to offer a comfortable construction of such points we make *Extended_homogeneous<RT>* a decorator/factory data type [GHJV95] for the geometric objects. Construction and conversion routines can be accessed as methods of the factory.

⟨*extended homogeneous*⟩≡

```
template <class RT_>
class Extended_homogeneous : public
  CGAL::Homogeneous< CGAL::RPolynomial<RT_> > { public:

  ⟨extended homogeneous kernel interface types⟩
  ⟨extended homogeneous kernel members⟩
};
```

We introduce the standard affine types into our kernel by prefixing them accordingly. The extended types carry the typenames without the prefix. Note that this decorator serves as a traits class to be used in algorithms that are based on our infimaximal frame. It is also the glue between the CGAL standard kernel and the extended geometric objects.

⟨*extended homogeneous kernel interface types*⟩≡

```
typedef CGAL::Homogeneous< CGAL::RPolynomial<RT_> >  Base;
typedef Extended_homogeneous<RT_> Self;

typedef CGAL::Homogeneous<RT_> Standard_kernel;
typedef RT_ Standard_RT;
typedef typename Standard_kernel::FT Standard_FT;
typedef typename Standard_kernel::Point_2     Standard_point_2;
typedef typename Standard_kernel::Segment_2   Standard_segment_2;
typedef typename Standard_kernel::Line_2      Standard_line_2;
typedef typename Standard_kernel::Direction_2 Standard_direction_2;
```

```
typedef typename Standard_kernel::Ray_2       Standard_ray_2;
typedef typename Standard_kernel::Aff_transformation_2
  Standard_aff_transformation_2;
typedef typename Base::RT RT;
typedef typename Base::Point_2      Point_2;
typedef typename Base::Segment_2    Segment_2;
typedef typename Base::Direction_2  Direction_2;
typedef typename Base::Line_2       Line_2;
// used only internally
enum Point_type { SWCORNER=1, LEFTFRAME, NWCORNER,
                  BOTTOMFRAME, STANDARD, TOPFRAME,
                  SECORNER, RIGHTFRAME, NECORNER };
```

We now implement the construction deduced above. For a non-standard point on the upper frame segment supported by a line $l \equiv ax+by+c = 0$ the polynomial coefficients are $m = -b/a, n = -c/a$. Accordingly on the left frame segment $m = -a/b, n = -c/b$.

⟨*non-standard point construction*⟩≡

```
Point_2 epoint(const Standard_RT& m1, const Standard_RT& n1,
               const Standard_RT& m2, const Standard_RT& n2,
                                      const Standard_RT& n3) const
{ return Point_2(RT(n1,m1),RT(n2,m2),RT(n3)); }
Point_2 construct_point(const Standard_line_2& l, Point_type& t) const
{
  t = (Point_type)Line_to_epoint<Standard_kernel>::determine_type(l);
  Point_2 res;
  switch (t) {
    case SWCORNER:   res = epoint(-1, 0, -1, 0, 1); break;
    case NWCORNER:   res = epoint(-1, 0,  1, 0, 1); break;
    case SECORNER:   res = epoint( 1, 0, -1, 0, 1); break;
    case NECORNER:   res = epoint( 1, 0,  1, 0, 1); break;
    case LEFTFRAME:
      res = epoint(-l.b(), 0,  l.a(), -l.c(), l.b()); break;
    case RIGHTFRAME:
      res = epoint( l.b(), 0, -l.a(), -l.c(), l.b()); break;
    case BOTTOMFRAME:
      res = epoint( l.b(), -l.c(), -l.a(), 0, l.a()); break;
    case TOPFRAME:
      res = epoint(-l.b(), -l.c(),  l.a(), 0, l.a()); break;
    default: CGAL_assertion_msg(0,"EPoint type not correct!");
  }
  return res;
}
```

### Type determination

To evaluate the results of an algorithm one also needs an operation that deduces the type from an epoint $p$. From the polynomial representation we can easily defer this type by checking the homo-

geneous components $p.hx(\,)$ and $p.hy(\,)$. Of course standard points have zero degree in both x- and y-components. For any non-standard $p$ on the frame we know that the relative interior of the frame box segments is specified by the condition that $|p.hx(\,)| \gtrless |p.hy(\,)|$. The sign of the larger component (larger with respect to its absolute value) determines the box segment. Equality $|p.hx(\,)| = |p.hy(\,)|$ specifies the corners of the box.

```
Point_type type(const Point_2& p)
{
  CGAL_assertion(p.hx().degree()>=0 && p.hy().degree()>=0 );
  CGAL_assertion(p.hw().degree()==0);
  if (p.hx().degree() == 0 && p.hy().degree() == 0)
    return STANDARD;
  // now we are on the square frame box
  RT rx = p.hx(), ry = p.hy();
  int sx = sign(rx), sy = sign(ry);
  if ( sx < 0 ) rx = -rx;
  if ( sy < 0 ) ry = -ry;
  if ( rx > ry )
    if (sx > 0) return RIGHTFRAME; else return LEFTFRAME;
  if ( rx < ry )
    if (sy > 0) return TOPFRAME; else return BOTTOMFRAME;
  // now (rx == ry)
  if ( sx == sy ) {
    if (sx < 0) return SWCORNER; else return NECORNER;
  } else { CGAL_assertion(sx==-sy);
    if (sx < 0) return NWCORNER; else return SECORNER;
  }
}
```

**Visualization**

We finally treat the problem of how to visualize extended objects. Given a set $S$ of extended points let us determine a concrete frame radius $R_0$ such that all standard points in $S$ are contained inside our frame but also all non-standard points in $S$ can be drawn on the correct frame box segments. Note that the latter is not trivially true for arbitrary small values of $R_0$.

Consider a line $l$ with slope $m$. If $|m| \neq 1$ the line $l$ intersects both angular bisectors of our coordinate frame. The intersection point of the larger absolute coordinates determines a lower bound for $R_0$. If $|m| = 1$ a natural lower bound for $R_0$ is half the length of the ordinate segment on the y-axis between $l$ and the origin. See Figure 3.3.

For our polynomial representation $(m_xR+n_x, m_yR+n_y, w)$ we know that for points in the interior of the frame box segments it holds that $|(m_xR+n_x)| \gtrless |(m_yR+n_y)|$. In either case we can set both polynomials equal and resolve for $R$ if $|m_x| \neq |m_y|$. $R = |(n_x-n_y)|/|(m_y-m_x)|$ presumed the line is not parallel to any of the angular bisectors of the coordinate frame. If $|m_x| = |m_y|$ then the constant parts $n_x/w$ or $n_y/w$ determine the abscissa or ordinate distances between the underlying line and the origin (depending on the frame segment that contains the extended point). At least one of $n_x/w$ or $n_y/w$ is actually zero (by definition of our extended points). In this case the minimum frame radius $R_0$ is half the absolute value of the abscissa or ordinate distance of the line to the origin.

We now code this determination of $R_0$ for an iterator range of extended points. Note that the common denominator of the homogenous representation is always a constant and positive. Note that

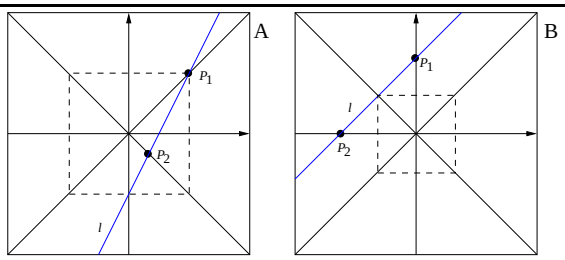



Figure 3.3: The point $P_1$ determines a lower bound for the frame radius $R_0$ to display the non-standard points at the tips of line $l$. In case (A) we take the absolute value of its coordinates, in case (B) we take half of its distance to the origin.

we round the integral division operations on the ring type up.

⟨*determining a lower bound for R*⟩≡
```
template <class Forward_iterator>
void determine_frame_radius(Forward_iterator start, Forward_iterator end,
                            Standard_RT& R0) const
{ Standard_RT R, mx, nx, my, ny;
  while ( start != end ) {
    Point_2 p = *start++;
    if ( is_standard(p) ) {
      R = max(abs(p.hx()[0])/p.hw()[0],
              abs(p.hy()[0])/p.hw()[0]);
    } else {
      RT rx = abs(p.hx()), ry = abs(p.hy());
      mx = ( rx.degree()>0 ? rx[1] : 0 ); nx = rx[0];
      my = ( ry.degree()>0 ? ry[1] : 0 ); ny = ry[0];
      if ( mx > my )      R = abs((ny-nx)/(mx-my));
      else if ( mx < my ) R = abs((nx-ny)/(my-mx));
      else /* mx == my */ R = abs(nx-ny)/(2*p.hw()[0]);
    }
    R0 = max(R+1,R0);
  }
}
```

### Extended predicates

Remember why the predicates *compare_xy*, *orientation*, *side_of_circle* are extensible. The first is just a cascaded comparison of coordinates (sign of their difference), the latter are sign-of-determinant calculations. The orientation predicate on three points is defined by the homogeneous expression:

$$\text{orientation}(p1,p2,p3) = \text{sign}\begin{vmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ w_1 & w_2 & w_3 \end{vmatrix}$$

Thus, evaluation of the sign means looking at the sign of the coefficient of $R$ if it is nonzero, or at the sign of the constant term if it is zero. The corresponding functionality is programmed into the sign

function of our polynomial ring number type *RPolynomial<NT>*. Thus adding the following methods to the extended geometry traits class implements the functionality via the kernel base class.

```
int compare_xy(const Point_2& p1, const Point_2& p2) const
{ typename Base::Compare_xy_2 _compare_xy = compare_xy_2_object();
  return _compare_xy(p1,p2);
}
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
{ typename Base::Orientation_2 _orientation = orientation_2_object();
  return _orientation(p1,p2,p3);
}
```

**Extended constructions**

Algorithms in computational geometry can be grouped into three categories: *subset selection*, *computation*, and *decision* [PS85, 1.4]. Algorithms of the first type resort to predicates, algorithms of the second type construct geometric objects. To cover this necessity software libraries like LEDA or CGAL offer a set of so called constructions in their geometric kernels. We have already shown that the intersection construction is extendible to be used with extended segments.

First we want to present three examples how the standard algebraic calculation of intersection points is blown up by common polynomial factors.

The coefficients of a line $l$ through two points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ are

$$a = y_1 - y_2, b = x_2 - x_1, c = x_1 y_2 - x_2 y_1. \tag{3.2}$$

The intersection point is defined by the common point of the two underlying lines $l_i \equiv (a_i x + b_i y + c_i = 0), i = 1, 2$. Their common point is then obtained by solving the linear system which has a solution if the lines are not parallel. We obtain

$$p_i = (b_1 c_2 - b_2 c_1, a_2 c_1 - a_1 c_2, a_1 b_2 - a_2 b_1) \tag{3.3}$$

in homogeneous representation. Apart from the formal argument why these quotients contain common factors and how they can be simplified to a minimal representation we give three examples.

**two non-standard points on one frame segment** — Look at the case where the frame segment is the upper one. Thus $p_i = (m_i R + n_i, R), i = 1, 2$. According to equation (3.2) we obtain

$$\begin{aligned} a &= R - R = 0 \\ b &= (m_2 - m_1)R + (n_2 - n_1) \\ c &= (m_1 - m_2)R^2 + (n_1 - n_2)R = bR \end{aligned}$$

The common factor is $b$, the underlying line is $l \equiv by + c = 0 \Leftrightarrow y - R = 0$. We obtain a simple parameterized version of a horizontal line supporting the upper frame segment. The three other cases are symmetric.

**two non-standard points spanning a standard line** — Look at the case of one point $p_1$ on the lower frame segment, $p_2$ on the upper frame segment, both on a line $l \equiv ax + by + c = 0$ where

we assume orientation from $p_1$ to $p_2$. We get according to Construction (3.1) $p_1 = (b/a\,R - c/a, -R)$, $p_2 = (-b/a\,R - c/a, R)$. According to equation (3.2) we obtain

$$\begin{aligned} a' &= -2R \\ b' &= -2R\,b/a \\ c' &= (b/a - b/a)\,R^2 - 2R\,c/a = -2R\,c/a \end{aligned}$$

The common factor is $-2R$. The underlying line is $l' \equiv a'x + b'y + c' = 0 \Leftrightarrow ax + by + c = 0$ as multiplying by $a$ and dividing by $-2R$ does not change the line.

**one non-standard point and one standard point spanning a standard ray** — We look again at a line $l \equiv ax + by + c = 0$ supporting $p_2$ on the upper frame segment and a standard point $p_1$ on this line. We have $p_1 = (b/a\,y_0 - c/a, -y_0)$, $p_2 = (-b/a\,R - c/a, R)$. According to equation (3.2) we obtain

$$\begin{aligned} a' &= (y_0 - R) \\ b' &= -b/a\,R - c/a + b/a\,y_0 + c/a = b/a\,(y_0 - R) \\ c' &= -b/a\,y_0 R - c/a\,R + b/a\,y_0 R + c/a\,y_0 = c/a\,(y_0 - R) \end{aligned}$$

The common factor is $(y_0 - R)$.

Note that the polynomial factors are very simple. The greatest common divisor operation and the polynomial division scheme of the *RPolynomial* data type can be used to do the simplificication.

```
void simplify(Point_2& p)
{ RT x=p.hx(), y=p.hy(), w=p.hw();
  RT common = x.is_zero() ? y : gcd(x,y);
  common = gcd(common,w);
  p = Point_2(x/common,y/common,w/common);
}
```

Now the intersection uses the kernel operation and simplifies the resulting point afterwards.

```
Point_2 intersection(
  const Segment_2& s1, const Segment_2& s2)
{ typename Base::Intersect_2 _intersect = intersect_2_object();
  typename Base::Construct_line_2 _line = construct_line_2_object();
  Point_2 p;
  CGAL::Object result = _intersect(_line(s1),_line(s2));
  if ( !CGAL::assign(p, result) )
  CGAL_assertion_msg(false,"intersection: no intersection.");
  simplify(p);
  return p;
}
```

We provide a similar kernel based on a *cartesian representation* of points. In this case we use *RPolynomial<NT>* fed with a field type and use standard polynomial division for simplification in the intersection construction. The latter replaces the *gcd* operation of the ring type in the homogeneous case.

### 3.4.3 Filtered implementation - Specialized kernel plus filtered inline

In this section we descibe a more advanced extended kernel than *Simple_extended_kernel*. This kernel tries to optimize runtime by use of a filter stage. It does not rely on polynomial arithmetic but on unrolled polynomial expressions directly programmed for the evaluation of the predicates and constructions. We explain the techniques used. We implement specialized kernel types that store the arbitrary precision coordinates but also intervals of number type double approximating them. Then we show how our running example, the orientation test, is implemented in its unrolled fashion. We finally present the intersection operation based on a case-dependent implementation. Our description covers all ideas needed to implement the whole extended kernel concept.

#### Kernel types

In contrast to the first approach based on a polynomial arithmetic type *RPolynomial* plugged into CGAL homogeneous points, we implement an extended kernel based on specialized types. The components are the types listed below plus the predicates and constructions that are required to construct a model of our *ExtendedKernelTraits_2* concept. The types that we implement are

```
SPolynomial<RT>
SQuotient<RT>
Extended_point<RT>
Extended_segment<RT>
Extended_direction<RT>
```

We shortly elaborate on the usage of the above types and their design. Let *RT* be a multiprecision integer number type like LEDA *integer*. *SPolynomial<RT>* is a container type storing linear polynomials of the form $mR+n$. *SQuotient<RT>* stores a two tuple consisting of an *SPolynomial<RT>* and an *RT* object and represents the corresponding quotient.

Our definition tells us that an extended point $p$ in homogeneous representation has the form $(m_xR+n_x, m_yR+n_y, w)$, where $R$ is our frame defining variable and all other identifiers are numbers from *RT*. *p.hx( )* returns the x-polynomial $m_xR+n_x$ and *p.hy( )* the y-polynomial $m_yR+n_y$ (of type *SPolynomial<RT>*). *p.hw( )* returns $w$. These are the homogeneous $x$- and $y$- coordinates of $p$ with common denominator $w$. For completeness $p$ also provides a Cartesian interface *p.x( )* returning an *SQuotient<RT>* of the form $(m_xR+n_x)/w$. In analogy $p.y( ) = (m_yR+n_y)/w$.

All number entries of $p$ can be accessed as multiprecision numbers as well as as double approximations stored in an interval of type *CGAL::Interval_nt_advanced*. Thus a point stores 5 *RT* entries and 10 double precision entries. The number type interface consists of the operations *p.mx( )*, *p.nx( )*, *p.my( )*, *p.ny( )*, *p.hw( )*, and *p.mxD( )*, *p.nxD( )*, *p.myD( )*, *p.nyD( )*, *p.hwD( )* for the intervals. The operation *p.is_standard( )* returns true, iff both $m_x$ and $m_y$ are zero.

The points are programmed along the lines of the LEDA and CGAL geometric kernel design. They have I/O stream operators, and they can be drawn in a LEDA window, when our frame parameter $R$ is fixed.

Points are realized by a smart-pointer scheme. There is a backend object type *Extended_point_rep<RT>* (the representation) and a frontend handle type *Extended_point<RT>*. We only elaborate on the representation type. Details on smart pointers are offered in the LEDA book [MN99].

⟨*extended points*⟩≡
```
template <typename RT> class Extended_point;
template <typename RT> class Extended_point_rep;

template <typename RT>
class Extended_point_rep : public Ref_counted {
  friend class Extended_point<RT>;
  SPolynomial<RT> x_,y_; RT w_;
  typedef Interval_nt_advanced DT;
  DT mxd,myd,nxd,nyd,wd;
public:
  ⟨construction⟩
};
```

For the filter stage we use interval approximations of type *CGAL*::*Interval_nt_advanced*. See H. Brönnimann et.al [BBP98] for more information. An object of this type is an interval of two doubles representing any number in the interval. An arithmetic operation *op* of $+$, $-$, $*$, $/$ on two intervals $X$ and $Y$ calculates an interval $X$ op $Y$ such that $\forall x \in X, y \in Y : (x \text{ op } y) \in (X \text{ op } Y)$. This allows us to determine the correct sign of an interval expression as long as the interval does not contain zero. The type uses exceptions to tell user code that a sign determination does not lead to a secure result. That exception can be catched to repair the resulting uncertainty. We will see how this works in our predicates below. The type *Interval_nt_advanced* implements dynamic filtering. Rounding errors are accumulated during the execution of the program. The type requires its user to take the responsibility for the rounding mode of the processor. Whenever an arithmetic interval expression is evaluated the processor should be in its correct rounding mode (switching the processor is an expensive operation, thereby the user's care does pay-off). The switching is done by class declaration statement *Protect_FPU_rounding*<*true*> *P*. The construction of object *P* sets the correct rounding mode, its destruction resets the previous mode which ensures correct execution of code parts that rely on different rounding modes.

The following conversion routine constructs an interval that contains a *double* approximation *cn* from its parameter *n* (a LEDA *integer*). Only two cases can occur: *n* can be approximated accurately by an interval $[cn, cn]$ of zero width if the bit representation of *n* has less than 53 bits. Otherwise we add the smallest representable *double* to make the interval contain *n*. By the addition the interval is expanded by exactly the radius of the machine accuracy. For more information on rounding problems please refer to D. Goldberg [Gol91].

```
DT to_interval(const leda_integer& n)
{ double cn = CGAL::to_double(n);
  leda_integer pn = ( n>0 ? n : -n);
  if ( pn.iszero() || log(pn) < 53 ) return DT(cn);
  else {
    Protect_FPU_rounding<true> P;
    return DT(cn)+CGAL::Interval_base::Smallest;
  }
}
```

On construction of the representation we construct the *double* approximation of the multiprecision entries. Note that we trade space for execution time.

⟨*construction*⟩≡
```
Extended_point_rep(const RT& x, const RT& y, const RT& w) :
  Ref_counted(), x_(x),y_(y),w_(w)
{ CGAL_assertion_msg(w!=0,"denominator is zero.");
  nxd=CGAL::to_interval(x);
  nyd=CGAL::to_interval(y);
  wd=CGAL::to_interval(w);
  mxd=myd=0;
}
Extended_point_rep(const SPolynomial<RT>& x,
                   const SPolynomial<RT>& y,
                   const RT& w) : Ref_counted(), x_(x),y_(y),w_(w)
{ CGAL_assertion_msg(w!=0,"denominator is zero.");
  mxd=CGAL::to_interval(x.m());
  myd=CGAL::to_interval(y.m());
  nxd=CGAL::to_interval(x.n());
  nyd=CGAL::to_interval(y.n());
  wd=CGAL::to_interval(w);
}
```

We do not show the implementation of the class *Extended_point*<*RT*>. It mainly serves as an interface of the representation class and inherits the handle maintainance code from the front end class *CGAL*::*Handle_for*.

## Predicates

We show the implementation of the orientation predicate of extended points. All other predicates follow the same strategy. For three homogeneous points in polynomials we derive the formula for the orientation determinant and build up a filter cascade. We implement three template functions that code the unrolled coefficients of the polynomial in *R* of degree 2. We do not prove the derivation of the following algebraic expressions, we instead explain how we obtained them. We used the math package Maple [CGG+91b] to do the algebra. The following script executed in maple produces the code expressions below. The lines with the comments have to be executed for the corresponding indices.

```
> pxi := mxi*R+nxi // i=1,2,3
> pyi := myi*R+nyi // i=1,2,3
> M := array([[px1,py1,w1],[px2,py2,w2],[px3,py3,w3);]]
> orient := collect(det(M),R);
> coeffi := coeff(orient,R,i); // i=0,1,2
> C(coeffi); // i=0,1,2
```

Finally just paste the coefficient code into the template functions. The following operations code the coefficient of the squared, linear, and constant term of the function in *R* that is the result of the determinant evaluation of *M*.

⟨*orientation predicate*⟩≡
```
template <typename NT> inline
int orientation_coeff2(const NT& mx1, const NT& /*nx1*/,
                       const NT& my1, const NT& /*ny1*/, const NT& w1,
                       const NT& mx2, const NT& /*nx2*/,
```

```
                        const NT& my2, const NT& /*ny2*/, const NT& w2,
                        const NT& mx3, const NT& /*nx3*/,
                        const NT& my3, const NT& /*ny3*/, const NT& w3)
{
  NT coeff2 = mx1*w3*my2-mx1*w2*my3+mx3*w2*my1-
              mx2*w3*my1-mx3*w1*my2+mx2*w1*my3;
  return CGAL_NTS sign(coeff2);
}
template <typename NT> inline
int orientation_coeff1(const NT& mx1, const NT& nx1,
                        const NT& my1, const NT& ny1, const NT& w1,
                        const NT& mx2, const NT& nx2,
                        const NT& my2, const NT& ny2, const NT& w2,
                        const NT& mx3, const NT& nx3,
                        const NT& my3, const NT& ny3, const NT& w3)
{
  NT coeff1 = mx1*w3*ny2-mx1*w2*ny3+nx1*w3*my2-mx2*w3*ny1-
              nx1*w2*my3+mx2*w1*ny3-nx2*w3*my1+mx3*w2*ny1+
              nx2*w1*my3-mx3*w1*ny2+nx3*w2*my1-nx3*w1*my2;
  return CGAL_NTS sign(coeff1);
}
template <typename NT> inline
int orientation_coeff0(const NT& /*mx1*/, const NT& nx1,
                        const NT& /*my1*/, const NT& ny1, const NT& w1,
                        const NT& /*mx2*/, const NT& nx2,
                        const NT& /*my2*/, const NT& ny2, const NT& w2,
                        const NT& /*mx3*/, const NT& nx3,
                        const NT& /*my3*/, const NT& ny3, const NT& w3)
{
  NT coeff0 = -nx2*w3*ny1+nx1*w3*ny2+nx2*w1*ny3-
               nx1*w2*ny3+nx3*w2*ny1-nx3*w1*ny2;
  return CGAL_NTS sign(coeff0);
}
```

Now the final orientation predicate consists of three *try-catch* blocks. Each *try* block contains the filtered coefficient evaluation. If the sign evaluation is not defined (the resulting interval contains zero) then the *unsafe_comparison* exception is thrown. The *catch* block evaluates the expression with *RT* arithmetic. Note again the protection of the rounding mode with the *Protect_FPU_rounding*<*true*> class declaration. The macros *INCTOTAL*( ) and *INCEXCEPTION*( ) are used to accumulate the statistics of the filter stages.

⟨*orientation predicate*⟩+≡

```
template <typename RT>
int orientation(const Extended_point<RT>& p1,
                const Extended_point<RT>& p2,
                const Extended_point<RT>& p3)
{
  int res;
  try { INCTOTAL(or2); Protect_FPU_rounding<true> Protection;
    res = orientation_coeff2(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
```

```
                              p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                              p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD());
    }
    catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(or2);
      res = orientation_coeff2(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                              p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                              p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw());
    }
    if ( res != 0 ) return res;
    try { INCTOTAL(or1); Protect_FPU_rounding<true> Protection;
      res = orientation_coeff1(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                              p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                              p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD());
    }
    catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(or1);
      res = orientation_coeff1(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                              p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                              p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw());
    }
    if ( res != 0 ) return res;
    try { INCTOTAL(or0); Protect_FPU_rounding<true> Protection;
      res = orientation_coeff0(p1.mxD(),p1.nxD(),p1.myD(),p1.nyD(),p1.hwD(),
                              p2.mxD(),p2.nxD(),p2.myD(),p2.nyD(),p2.hwD(),
                              p3.mxD(),p3.nxD(),p3.myD(),p3.nyD(),p3.hwD());
    }
    catch (Interval_nt_advanced::unsafe_comparison) { INCEXCEPTION(or0);
      res = orientation_coeff0(p1.mx(),p1.nx(),p1.my(),p1.ny(),p1.hw(),
                              p2.mx(),p2.nx(),p2.my(),p2.ny(),p2.hw(),
                              p3.mx(),p3.nx(),p3.my(),p3.ny(),p3.hw());
    }
    return res;
  }
```

Corresponding implementations are provided for the predicates *compare_x*( ), *compare_y*( ), *compare_xy*( ), and *compare_pair_dist*( ). The latter realizes the squared distance comparison of two pairs of points. The resulting polynomial has again degree 2 but contains more complicated expressions than the orientation predicate above.

### Extended segments and intersection

We provide extended segments and some primitives. We mostly concentrate on the intersection predicate of lines supported by non-trivial extended segments. In this implementation we want to avoid the calculation of the polynomial *gcd*. Therefore, we extract the possible intersection configurations in advance and avoid higher degree polynomials resulting from the general algebraic term.

⟨*extended segments*⟩≡

```
  template <typename RT>
  class Extended_segment {
    Extended_point<RT> _p1,_p2;
  public:
    Extended_segment() : _p1(),_p2() {}
```

```
  Extended_segment(const Extended_point<RT>& p1,
                   const Extended_point<RT>& p2) :
    _p1(p1), _p2(p2) {}
  Extended_segment(const Extended_segment<RT>& s) :
    _p1(s._p1), _p2(s._p2) {}
  Extended_segment<RT>& operator=(const Extended_segment<RT>& s)
  { _p1 = s._p1; _p2 = s._p2; return *this; }
  const Extended_point<RT>& source() const { return _p1; }
  const Extended_point<RT>& target() const { return _p2; }
  void line_equation(RT& a, RT& b, SPolynomial<RT>& c) const;
};
```

We extract the line equation $ax+by+c=0$ directly from non-standard points if the extended segment is just a segment. Note that for all lines crossing the interior of our box $a,b,c$ are just constants from our integer ring type $RT$. However the frame segments also support lines of the from $x\pm R=0$ and $y\pm R=0$. Therefore, we allow a linear polynomial for the coefficient $c$.

⟨*extended segment primitives*⟩≡

```
template <typename RT>
void Extended_segment<RT>::
line_equation(RT& a, RT& b, SPolynomial<RT>& c) const
{
  bool sstandard = _p1.is_standard();
  bool tstandard = _p2.is_standard();
  if (sstandard && tstandard) {
    ⟨standard segment⟩
  }
  Extended_point<RT> p;
  bool correct_orientation=true;
  if (!sstandard && !tstandard) {
    ⟨two points on the frame box⟩
  }
  else if (sstandard && !tstandard)
  { p = _p2; }
  else if (!sstandard && tstandard)
  { p = _p1; correct_orientation=false; }
  ⟨one point on the frame box⟩
}
```

If two points span a standard segment then the three coefficients of the line through the points can be derived from the following determinant equation (just resolve for the variables in the last row).

$$\begin{vmatrix} x1 & y1 & w1 \\ x2 & y2 & w2 \\ a & b & c \end{vmatrix} = 0$$

.

⟨*standard segment*⟩≡
```
a = _p1.ny()*_p2.hw() - _p2.ny()*_p1.hw();
b = _p1.hw()*_p2.nx() - _p2.hw()*_p1.nx();
c = SPolynomial<RT>(_p1.nx()*_p2.ny() - _p2.nx()*_p1.ny());
return;
```

Two points on the box produce two basic configurations. Either the points are both on one frame segment. Or they are part of one affine line crossing the box. If they lie on the same frame segment then their corresponding coordinate polynomials are equal in homogeneous representation and equal to $\pm R$. Note that we keep the algebraic calculation within the polynomials of degree less than 2. When both points lie on the same frame segment then the minimal representation of the corresponding coordinate polynomial is $\pm 1R + 0$. This leads to line equations of the form $0x + 1y + \mp R = 0$ when the y-coordinates are equal and to $1x + 0y + \mp R = 0$ when the x-coordinates are equal. In case that the points span a standard affine line we forward the treatment to the mixed case by setting $p$ to one of the points.

⟨*two points on the frame box*⟩≡
```
bool x_equal = (_p1.hx()*_p2.hw() - _p2.hx()*_p1.hw()).is_zero();
bool y_equal = (_p1.hy()*_p2.hw() - _p2.hy()*_p1.hw()).is_zero();
if (x_equal && CGAL_NTS abs(_p1.mx())==_p1.hw() && _p1.nx()==0 )
{ int dy = (_p2.hy()-_p1.hy()).sign();
  a=-dy; b=0; c = SPolynomial<RT>(dy*_p1.hx().sign(),0); return; }
if (y_equal && CGAL_NTS abs(_p1.my())==_p1.hw() && _p1.ny()==0 )
{ int dx = (_p2.hx()-_p1.hx()).sign();
  a=0; b=dx; c = SPolynomial<RT>(-dx*_p1.hy().sign(),0); return; }
p = _p2; // evaluation according to mixed case
```

Finally we have the point $p$ at the tip of a line (somewhere on the frame box). Obviously we can extract the affine equation of the line from the polynomial representation. We just set the parameter $R$ to 0 and 1, obtain two points, and calculate the equation. Note that by the special structure of our non-standard points the $2 \times 2$ determinants reduce nicely. $p.hw(\,)$ is a common factor of all coefficients $a, b, c$ of the line:

$$a = y_1 w_1 - y_2 w_1, b = x_2 w_1 - x_1 w_2, c = x_1 y_2 - x_2 y_1$$

where

$$x_1 = p.nx(\,), y_1 = p.ny(\,), w_1 = p.hw(\,), x_2 = p.mx(\,) + p.nx(\,), y_2 = p.my(\,) + p.ny(\,), w_2 = p.hw(\,)$$

For $a$ and $b$ it is obvious that $p.hw(\,)$ can be canceled out, and the difference simplified. $c$ can be reduced to $p.nx(\,) * p.my(\,) - p.ny(\,) * p.mx(\,)$ due to the special choice of our points. Remember that for non-standard points either their x- or y-coordinate is equal to $\pm R$. In the homogeneous representation this means that either $p.mx(\,) = \pm p.hw(\,)$ and $p.nx(\,) = 0$, or $p.my(\,) = \pm p.hw(\,)$ and $p.ny(\,) = 0$. In either case one can safely divide by $p.hw(\,)$.

⟨*one point on the frame box*⟩≡
```
RT x1 = p.nx(), y1 = p.ny();                // R==0
RT x2 = p.mx()+p.nx(), y2 = p.my()+p.ny(); // R==1
RT w = p.hw();
RT ci;
```

```
if ( correct_orientation ) {
  a = -p.my(); // (y1*w-w*y2)/w
  b =  p.mx(); // (x2*w-w*x1)/w
  ci = (p.nx()*p.my()-p.ny()*p.mx())/w; // (x1*y2-x2*y1)/w;
} else {
  a =  p.my(); // (y2*w-w*y1)
  b = -p.mx(); // (x1*w-w*x2)
  ci = (p.ny()*p.mx()-p.nx()*p.my())/w; // (x2*y1-x1*y2)/w;
}
c = SPolynomial<RT>(ci);
```

We finally provide the intersection construction of two lines supported by two segments. We use the linear system that defines the common point of two lines.

$$\begin{pmatrix} a1 & b1 \\ a2 & b2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -c_1 \\ -c_2 \end{pmatrix}$$

Note that the line equations are either standard affine lines or lines supporting box segments. The expressions for $x$ and $y$ are polynomials in $R$ up to degree 1, for $w$ it is just a constant.

⟨*extended segment primitives*⟩+≡

```
template <typename RT>
Extended_point<RT> intersection(
  const Extended_segment<RT>& s1, const Extended_segment<RT>& s2)
{
  RT a1,b1,a2,b2;
  SPolynomial<RT> c1,c2;
  s1.line_equation(a1,b1,c1);
  s2.line_equation(a2,b2,c2);
  SPolynomial<RT> x = c2*b1 - c1*b2;
  SPolynomial<RT> y = c1*a2 - c2*a1;
  RT w = a1*b2 - a2*b1; CGAL_assertion(w!=0);
  ⟨reduce point representation by gcd operation⟩
  return Extended_point<RT>(x,y,w);
}
```

We introduce an option that allows us to reduce the homogeneous representation of points by dividing both polynomials by the gcd of their content[10] and their common denominator. This leads to a representation of minimal bitlength. We found that this reduction pays off a lot. Remember that we use the kernel in binary operations of Nef polyhedra and their recursive usage of intermediate structures accumulates long point representations without this reduction.

⟨*reduce point representation by gcd operation*⟩≡

```
#ifdef REDUCE_INTERSECTION_POINTS
RT xgcd,ygcd;
if ( x.m() == RT(0) )  xgcd = ( x.n() == 0 ? RT(1) : x.n() );
else /* != 0 */    xgcd = ( x.n() == 0 ? x.m() : gcd(x.m(),x.n()) );
if ( y.m() == RT(0) )  ygcd = ( y.n() == 0 ? RT(1) : y.n() );
else /* != 0 */    ygcd = ( y.n() == 0 ? y.m() : gcd(y.m(),y.n()) );
```

[10] Remember: the content of a polynomial is the gcd of all nonzero coeffi cients.

```
RT d = gcd(w,gcd(xgcd,ygcd));
x /= d;
y /= d;
w /= d;
#endif // REDUCE_INTERSECTION_POINTS
```

### The kernel wrapper

All operations are either mapped to the above primitives or similar to the ones in the default homogeneous kernel. We do not present the layout of the kernel class *Filtered_extended_homogeneous*<*RT*>. It is similar to *Extended_homogeneous*<*RT*>. We only present the usefulness of *Extended_homogeneous*<*RT*> with respect to checking of our advanced kernel. We used the naive approach to back-up the results of our filtered code. As the expressions in this code base are much more complicated, errors were hard to determine. We enriched the filtered kernel by checking statements that we switched on when runtime examples seemed to produce problems. (Of course we had to log our random test inputs to allow checking in case of errors). When switched on by using the compile flag *KERNEL_CHECK* our checking macro is defined to be

```
#define CHECK(c1,c2) CGAL_assertion((c1) == (c2));
```

Then the orientation member of *Filtered_extended_homogeneous*<*RT*> is just defined as:

```
int orientation(const Point_2& p1, const Point_2& p2, const Point_2& p3)
const
{ CHECK(K.orientation(p1.checkrep(),p2.checkrep(),p3.checkrep()),
        CGAL::orientation(p1,p2,p3))
  return CGAL::orientation(p1,p2,p3); }
```

where *Point_2* is the extended point type in the local scope of *Filtered_extended_homogeneous*<*RT*>. *p.checkrep*( ) returns an extended point of type *Extended_homogeneous*<*RT*>::*Point_2* based on the *RPolynomial*<*RT*> number type. And *K* is a kernel object of type *Extended_homogeneous*<*RT*> in the local scope.

At last the filtered kernel has a member method *print_statistics*( ) that outputs the total number of failed filter stages in its base version. If the kernel is used with the compile flag *KERNEL_ANALYSIS* then each filtered code section is evaluated separately. All sections give the number of failed stages and the number of total evaluations thereby the efficiency of the filter can be evaluated with respect to the input used. This ends the description of the filtered extended kernel model.

## 3.5 Conclusions

In this chapter we have introduced infimaximal frames, a concept that allows to unify the treatment of planar rectilinear subdivisons. We have implemented three flavors of a extended kernel concept that can be used in application programms that wrap around the kernel concept.

We describe two applications, the first one being a subtaks of our second. The first application is the computation of arrangements of segments, rays, and lines. We can apply the generic segment sweep algorithm as described in Section 4.8 together with epoints and esegments. We have provided a LEDA extension package [LEDb] based on a kernel similar to *Filtered_extended_kernel* except that it uses static filtering instead of the CGAL interval arithmetic. The package contains geometric objects

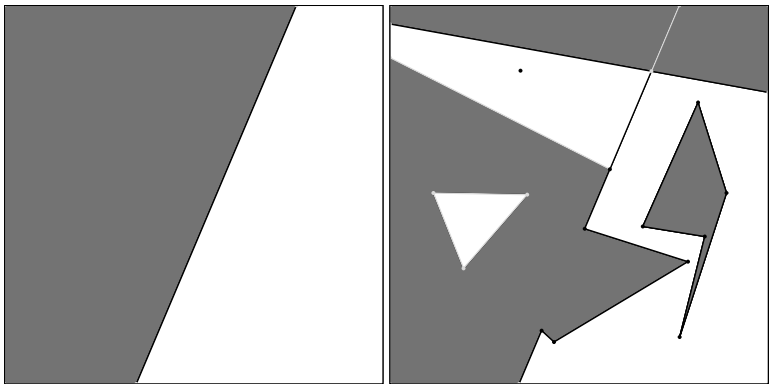


Figure 3.4: The left part shows a half-space pruned in the frame. The right part shows a complicated Nef polyhedron consisting of diverse faces and low dimensional features (a ray taken from a face and an isolated vertex). All vertices are embedded via extended points. All points on the square boundary are non-standard points.

and primitives similar to the affine two-dimensional LEDA kernel and sweep methods that work for rays, lines and extended segments.

The second application is an implementation of planar Nef polyhedra; in fact, this application made us think about the problem of making rays and lines look like segments. Figure 3.4 shows two Nef polyhedra. We view Nef polyhedra as embedded into an infimaximal frame. This makes all faces (except for the outside of the frame) bounded and allows us to represent planar Nef polyhedra by selective plane maps. The position of a vertex is given by an epoint and all edges of a Nef polyhedron correspond to esegments. All faces have circular closed face cycles.

In the following chapter we implement an overlay engine for plane maps that works for both the bounded affine scenario as well as the unbounded Nef structure. The engine is based on a generic sweep algorithm; instantiating it with an affine geometry kernel (*e.g.*, LEDA's or CGAL's) makes it work for bounded maps and instantiating it with extended points and segments makes it work for Nef polyhedra. We postpone runtime experiments of our extended kernel types to the end of Chapter 4.

# Chapter 4

# Planar Nef Polyhedra

## 4.1 Motivation

Nef polyhedra are the most general model for rectilinearly bounded subsets of affine space. Their definition is surprisingly simple whereas the operations that are supported without leaving the model are versatile. Nef's model of polyhedra does not impose topological restrictions on the sets that can be modeled like manifold or regularized models do. This implies of course that the abstract representation of the underlying theory has to cope with general topological complexity. The main reason for us to offer a data type Nef polyhedron is that many other models that are standard concepts in the field are covered by Nef's model:

- A *convex polytope* is defined as the convex hull of a nonempty finite set of points. Convex polytopes are thus compact closed and manifold sets. [Grü67]
- An *elementary polyhedron* is defined as the union of a finite number of convex polytopes. [Grü67]
- A *polyhedral set* is defined as the intersection of a finite number of closed half-spaces. Such sets are closed and convex but need not to be compact. [Grü67]
- The set of all points belonging to the simplices of a *simplicial complex* is normally called a (rectilinear) polyhedron. [Lef71]

This list shows that a system modeling Nef polyhedra enables a user to calculate in many interesting domains.

## 4.2 Previous Work

We cite the main publications from the field and present its development. We will first give an outline, then we deepen the notions that are interesting with respect to our research. Other notions are solely linked to the literature.

The theory of Nef polyhedra was first published in W. Nef's book "Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergraphik" [Nef78]. The book presents a mathematically sound theory of a general kind of polyhedra in arbitrary dimension and provides a great intuition about the generality of the elaborated concepts. The algorithmic part does not specialize in dimension. It should be clear that by only realizing a fixed low-dimensional data type the corresponding algorithms and data structures can be streamlined in runtime and space requirements.

In the following considerations we concentrate mainly on the presented data structure and the realization of a binary set intersection operation which poses the most requirements[1] on the underlying data structures and algorithmic modules.

On the workshop on computational geometry in Würzburg [BN88] a refined elaboration of the book concepts was given. The paper introduces the later so-called *Würzburg structure* which is the set of all low-dimensional[2] faces of a polyhedron. Each such face is stored in form of its local pyramid. Thus, the data structure is essentially a collection of pyramids. Each pyramid is realized by a selective arrangement of hyperplanes. This representation of pyramids is *not lean* in low dimensions and can be improved. Moreover, no incidence relation is coded into the collection. The intersection operation of two polyhedra is defined on top of this structure and is based on two techniques: recursion in dimension and superposition of two local pyramids. Neither space nor runtime bounds are given for the algorithmic description which apart from that is clearly structured. The paper solves some subproblems by introducing a symbolic parameter to obtain a symbolic hyperplane at infinity. However the transfer from the affine to the symbolic objects is part of the algorithmic flow and not encapsulated into a geometric kernel as we proceed in Chapter 3.

H. Bieri's introduction [Bie95] was a step to market Nef polyhedra to a wider audience. It provides a summary of the book and introduces all notions in a more tutorial-style picture. It also describes a small test system realizing binary set and simple topological operations written in PASCAL based on the so-called *extended Würzburg structure*. The predicate "extended" stems from the addition of incidence links to the Würzburg structure[3].

The article [Bie94] describes the realization of simple topological and set operations on top of the Würzburg structure. It has an introductory part and an algorithmic part. The key operation of interest, the intersection of two Nef polyhedra, is defined in pseudo code in a dimension recursive manner (up to dimension three but a possible generalization to higher dimensions is sketched). The main phase of the operation is based on a spatial sweep approach but the presentation is rather condensed, mainly mathematical, and lacks runtime and space bounds. In fact, the described procedure up to dimension two is quadratic and we will improve this bound to the optimal plane sweep bound (of segment intersection).

A follow up to the previous publication is the article [Bie96] that mainly closes open details of algorithmic considerations of the previous articles. Its main impact is the introduction of the *reduced Würzburg structure*. H. Bieri shows that it suffices to store the pyramids[4] of faces that are minimal elements of the incidence relation (a partial order defined via closure relation). As a consequence this reduces the space requirements of the pyramid collection but requires additional algorithmic processing when collecting all faces. The proposition has a strong impact on the representation of Nef polyhedra when we consider dimension three or higher. In space the above result implies that the pyramids of all vertices suffice to completely describe a compact polyhedron.

[1] The calculation of closure, interior, or complement is much simpler due to the fact that the faces of the input polyhedron are part of the faces of the output polyhedron in these cases.

[2] not full-dimensional

[3] However, the extension is only based on an untyped list.

[4] Pyramids represent faces.

The paper [Bie98] embeds Nef polyhedra in the field of solid modeling by offering conversion routines between previously defined data structures for Nef polyhedra: the reduced Würzburg structure, selective cellular complexes based on hyperplane arrangements, CSG-trees based on half-spaces, and binary space partitions. Again there are no time and space bounds.

J.R. Rossignac and M.A. O'Connor [RO90] have introduced *Selective Geometric Complexes* (SGC). An SGC consists of a cellular complex (the topological structure) and the corresponding geometric support spaces. Geometrically SGCs are pretty general. They use *real algebraic varieties* as the geometric elements that support cells. Such varieties can be decomposed into finite sets of connected smooth manifolds (so-called *extents*). An SGC is therefore a collection of mutually disjoint cells such that (1) each cell is a relatively open subset of an extent, (2) for each cell of the complex its boundary (a set of cells) is also part of the complex, and (3) each cell has a Boolean selection flag. The dimension of a cell is determined by the dimension of its underlying extent. The point set modeled by such a complex is the union of all selected cells.

One important concept is the incidence relation on cells. In SGCs it is defined in terms of a *boundary* and a *star* relation stemming from the corresponding concepts of simplicial complexes. Moreover due to the possibly curved geometry of extents the notion of *neighborhood* (orientation) is introduced to disambiguate degenerate boundary conditions.

The description of SGCs is still abstract and leaves room for refinement concerning the realization of the necessary data structure concepts[5]. The paper presents the central ideas and abstract definition of SGCs and its notions. It sketches binary operations based on the data type separated into phases. (We use this approach later in our implementation). On the other hand the paper omits many concrete considerations of the algorithmic subtasks (boundary evaluations, merging complexes, etc.) including runtime and space complexity. The problem of unbounded structures is not an issue.

How do SGCs relate to Nef polyhedra? The support spaces of Nef polyhedra are flats. Therefore, varieties and extents are no separate concepts. Many geometric ambiguities do not occur. The theory of Nef polyhedra as described by Nef and Bieri varies in the way how the exterior of Nef polyhedra is modeled. In their later papers the exterior is a *non-proper* face and thereby the ambient space is completely partitioned. With SGCs the exterior is no cell. We want to stress the following similarity. Assume we realize SGCs geometrically restricted to flats. Then, the simplification algorithm as part of the algorithmic description of the binary operations on SGCs produces cells that are the connected components of proper Nef faces. (The simplification algorithm is described and used in Section 4.7 of this thesis).

K. Dobrindt, K. Mehlhorn, and M. Yvinec [DMY93] describe an efficient algorithm for the intersection of a convex polyhedron and a Nef polyhedron in three-dimensional space. The presentation uses a *local graph* data structure modeling the local view that we introduce below. The local graph data structure is used as a vehicle to project the three dimensional topological neighborhood (the local pyramid) that defines Nef facets into the surface of a sphere centered at a point of interest. Their idea greatly simplifies the representation of local pyramids in three dimensions and allows a space efficient representation thereof (linear as opposed to the possible quadratic space of the original proposed arrangements). The corresponding algorithm was not implemented.

K. Mehlhorn and S. Naeher have introduced the notion of planar *generalized* polygons and implemented them in LEDA [MN99, Section 10.8]. A generalized polygon is a point set bounded by possibly several (weakly) simple polygonal chains. This topological restriction implies that generalized polygons are the same as regularized compact Nef polyhedra.

[5] A promised follow-up paper never appeared.

V. Ferrucci [Fer95] presented two implementation efforts to realize a data type modeling Nef polyhedra. His first approach is based on selective simplicial complexes and restricts the model to bounded Nef polyhedra. All simplices are rectilinearly embedded into the affine subspace spanned by their vertices and represent relatively open convex sets. All simplices (including subsimplices) of the complex are selectable by a Boolean flag. The point set of such a simplicial complex is the union of the embeddings of the selected simplices. In this approach Nef faces are only present implicitly as a union of simplices. The simplicial complex is thus a conforming simplicial subdivision of the bounded Nef polyhedron. In the second part of the paper Ferrucci proves that binary space partitions can be used to realize general Nef polyhedra. Both representations do not provide runtime or space qualification.

Several algorithmic descriptions from the above list either assume general position of their inputs to avoid degeneracies or even require what is called regular intersection. In some cases, the generally present robustness problems are tackled by transformations of the underlying coordinate system to avoid degenerate inputs and minimize robustness problems. The possibility of that approach was presented in [NS90].

Our approach differs from the previous work in several aspects. We elaborate on the planar case which is of course easier than the higher-dimensional case but leaves room for optimization via specialization. We solve the problem of degeneracy and robustness. We use standard data structures of our field to represent Nef polyhedra. We generalize an optimized plane sweep framework that can handle all degenerate cases for the binary operations and meet the optimal time and space bounds. One of our main contributions is the introduction of an extended geometric kernel that encapsulates the necessary geometric predicates to run the operations of the data type. In Chapter 3 we show the implementation of the kernel in two flavors, one which is simple to implement and one which is tuned by filter methods and is both robust and fast.

Quite some effort is put into the user interface of the software. Our data type offers the user to get her hand on faces by handles and explore the incidence of an object within the geometric structure. Our data type is based on a space efficient implementation of plane maps. It incorporates an intuitive exploration interface and allows further attribution of the objects. Our data type could be considered a flavor of the extended Würzburg structure where we shift the incidence into the center of our attention. Faces are not represented by their pyramids but the pyramids can be infered from the incidence relation and the extent of any face can be explored via incident lower-dimensional faces at a cost linear in its size. It is our conviction that the original Würzburg design is reasonable in higher-dimensions but means a loss of strength in the planar case. We also add functionality. Exploration of a geometric complex needs point location and ray shooting operations to link a user's geometric question into the geometric complex and its incidence. Finally our approach unifies the handling of special cases. By the introduction of infimaximal frames (Chapter 3) and their integration into the geometric complex we enclose the geometric complex into a symbolic box. Exploration and maintainance of the structures become much easier and more homogeneous as the faces of minimal dimension are always vertices. The latter has interesting consequences as Bieri has shown that the local pyramids of minimal faces describe the polyhedron completely.

In this project we realize a data type *Nef_polyhedron_2*. We present a software project clearly separated into different modules responsible for the different aspects of its realization. We will start with the theory, derive an abstract representation, map it to an abstract data type and add the algorithmic components.

In Section 4.3 we present the abstract knowledge about Nef polyhedra and introduce the notions that we use. Afterwards, in Section 4.4 we present the necessary software components of our design. We introduce the geometric and topological modules and their interaction. Then, in Section 4.5 we

present the concrete software design of our main interface data type and describe the interaction of different modules to implement the geometric methods of that data type. In Section 4.5 we describe the top-level software design, in Section 4.6 we provide some basics how we integrated the CGAL HDS as our plane map data type. In Section 4.7 we describe the detailed techniques that implement overlays of segments and plane maps. Afterwards, in Section 4.8 we present a generic plane sweep framework used as a working horse in the former implementation. Finally in Section 4.9 we present runtime results and further applications.

## 4.3 The Theory

We start with the formal definition of Nef polyhedra.

**Definition 6 (Nef Polyhedra [Nef78]):** A set $P \subseteq \mathcal{R}^n$ is a *Nef polyhedron* if $P$ is the result of a recursive application of set intersection and set complement starting from open half-spaces.

This definition supports the claim that they are the most general framework to handle polyhedral sets. As set union, set difference, and symmetric set difference can be reduced to intersection and complement all these set operations are closed in the model.

H. Bieri later gave alternative definitions for Nef polyhedra and proved their equivalence.

**Fact 1 (Bieri [Bie95]):** The original definition is equivalent to any of the following conditions

1. $P$ corresponds to the root of a CSG-tree with closed half-spaces as leaf primitives and intersection, union and difference as internal nodes.
2. There exist two finite families $F = \{f_1, \ldots, f_n\}$ and $G = \{g_1, \ldots, g_m\}$ of relatively open subsets of $\mathcal{R}^n$ such that $P = \bigcup_i f_i$ and $\mathsf{cpl}\,P = \bigcup_j g_j$.
3. There exists a set of hyperplanes $H$ such that $P$ is the union of some cells of the arrangement $\mathcal{A}(H)$.

(1) gives a link to constructive solid geometry. (2) links Nef polyhedra to cellular complexes and (3) to hyperplane arrangements. When studying the original theory actually the third equivalence is the key observation. Many propositions about the point set $P$ can be reduced to an examination of the minimal building blocks of the polyhedron: the cells of the arrangement built by the hyperplanes that define the polyhedron.

The elegance of the definition is carried forward to the notion of faces.

**Definition 7 (Local pyramids and Faces):** Let $K \subseteq \mathcal{R}^n, x \in \mathcal{R}^n$. We call $K$ a *cone with apex* 0 if $K = \mathcal{R}^+ K$ and cone with apex $x$ if $K = x + \mathcal{R}^+(K - x)$. A cone which is also a polyhedron is called a *pyramid*.

Now let $P \subseteq \mathcal{R}^n$ be a polyhedron and $x \in \mathcal{R}^n$. There is a neighborhood $U_0(x)$ such that the pyramid $Q := x + \mathcal{R}^+((P \cap U(x)) - x)$ is the same for all neighborhoods $U(x) \subseteq U_0(x)$. $Q$ is called the *local pyramid* of $P$ in $x$ and denoted $P^x$.

A *face* $s$ of $P$ is then a maximal non-empty subset of $\mathcal{R}^n$ such that all of its points have the same local pyramid $Q$, i.e., $s = \{x \in \mathcal{R}^n : P^x = Q\}$. In this case $Q$ is also denoted $P^s$. The dimensions of a face is the dimension of its affine hull $\mathsf{dim}(s) := \mathsf{dim}(\mathsf{aff}\,s)$.

Note that this notion of a face partitions $\mathcal{R}^n$ into faces of different dimension. Faces as defined by Nef do not have to be connected. There are only two full-dimensional faces possible whose local pyramids are the space itself or the empty set. All lower-dimensional faces form the *boundary* of the

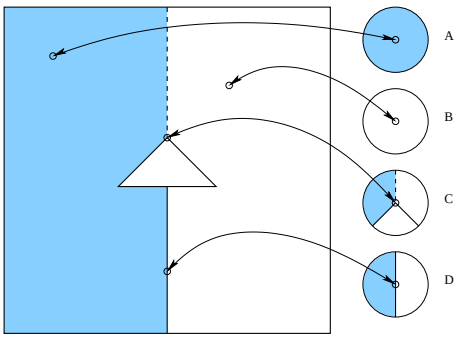



Figure 4.1: The polyhedron $P$ consists of the colored face, the triangle boundary and the vertical segment below the triangle. Some local views of $P$ are: (A) the full plane, (B) the empty set, (C) a radial cake of sectors, (D) a half-space including its boundary.

polyhedron. As usual we call zero-dimensional faces *vertices* and one-dimensional faces *edges*. In the plane we call the full-dimensional faces *2-faces* or just *faces* when the meaning is clear from the context.

**Definition 8 (Incidence):** Two faces $s$ and $t$ (in their general sense) are *incident* if $s \subseteq \mathsf{clos}\, t$.

We will treat incidence as a bidirectional relation. We say that $s$ is *downward-incident* to $t$ and conversely that $t$ is *upward-incident* to $s$.

We now list some facts about faces. The proofs for these facts can be found in Nef's book [Nef78]. We append the chapter and theorem numbers separated by a semicolon. All faces of a polyhedron are polyhedra [6;1.1]. Faces are relatively open sets [6;2]. The linear subspace of all apices of the pyramid associated with a face is the affine hull of the face [6;2]. $x \in P^x$ iff $x \in P$ [3;7]. Let $s$ be a face of the polyhedron $P$ and let $t$ be a face of $s$. Then, $t$ is the union of some faces of $P$ [6;12]. A face of $P$ is either a subset of $P$ or disjoint from $P$ [6;4]. For two faces $s$ and $t$ of $P$ either $s \subseteq \mathsf{clos}\,(t)$ or $s \cap \mathsf{clos}\,(t) = \emptyset$ [6;9,10].

Remember that Nef edges and Nef 2-faces are not necessarily connected. Note also that some connected components of edges are not necessarily bounded by a vertex. How do the local pyramids of any point $x$ in the plane look like? We can represent them by the intersection of a small enough neighborhood disc centered at $x$ with its pyramid $P^x$. The disc is partitioned into sectors by radial segments. We assign a mark to the center, to any radial segment, and to the sectors in between the radial segments such that the corresponding point set is marked if it belongs to the local pyramid of $x$. We also call such a disc together with its marks the *local view* of $x$ (cf. Figure 4.1).

**Lemma 4.3.1:** The local view of a point $x$ has the following properties:

1. the mark of any radial segment is different from one of the two sectors incident to it.
2. $x$ is contained in a 2-face of $P$ iff the local view is a disc that contains no radial segments and is marked as the center.
3. $x$ is contained in an edge of $P$ iff the local view contains exactly two radial segments that are part of a line through the center; the center and the two radial segments are marked equally but their common mark is different from at least one sector of the disc.

4. $x$ is a vertex of $P$ iff the mark of the center is different from at least one radial segment or one sector of the disc and the local view is not that of item 3.

*Proof.* (1) follows from the fact that radial segments are part of the boundary of an open or closed half-plane and a point on this boundary has the local view of that half-plane. (2) refers to the two trivial pyramids: the empty set and the full plane. (3) follows from the fact that edges are part of the boundary of open and closed half-spaces. (4) if all marks are equal then $x$ would have a totally marked or non-marked disk neighborhood. But that's the neighborhood of a 2-face. □

To determine the local view of a point $x$ with respect to a polyhedron $P$ is called to *qualify* $x$ with respect to $P$.

## 4.4 The Data Structure

We want to store Nef polyhedra in an intuitive way and want to use generally known data structures. Faces should be objects whose extent and topological neighborhood (incidence) can be explored. We revert the role of pyramids and incidence of the extended Würzburg structure. In our approach incidence is the key concept, pyramids can be derived from it.

Starting from the last item of Fact 1 we have to model (parts) of an arrangement of lines in the plane. Moreover we want to model the Nef faces including their incidences. 2-faces of a polyhedron are in general two dimensional sets of points bounded by chains of segments that do not have to be simple, can be circularly closed, or open. The latter happens when 2-faces extend to infinity. Then the chain of segments has rays as its first and last element. A simple line bounding a 2-face can be seen as two oppositely oriented rays starting in the same point.

Therefore, concerning the geometry of edges we need to model straight line objects like segments, rays and lines, which are the result of intersection operations of half-spaces. To simplify the treatment of unbounded structures we use the concept of extended points and infimaximal frames which we have formally introduce in Chapter 3 and which builds the geometric layer of our design. Consider an axis-parallel squared box centered at the origin. Any ray is pruned by this box in a so-called non-standard point (corresponding to the ray-tip). The assignment of ray-tips to box segments becomes topologically constant when we grow the framing box above a certain size. The frame is called infimaximal because it is always large enough to enclose all concrete geometric objects like points and segments in its interior that appear in the execution of our algorithms. Extended points are defined to be standard points and the non-standard points corresponding to ray-tips. They allow us to represent segments, rays and lines by a pair of such points and we get rid of the infinite extent of the unbounded structures. Adding such a frame to bound the plane, all unbounded faces become symbolically bounded structures, their boundary becomes a cyclic structure.

To realize Nef faces including their incidence relations we need a cellular subdivision of the plane. We use a plane map data type (bidirected, embedded graphs). The incidence relations between the objects of a plane map (vertices, edges, and faces) reflect the topology of the local pyramids of the planar Nef polyhedron. The plane map concept is presented in the introduction of our notions and is our topological bottom layer. Most newer textbook recommend the use of doubly connected edge lists (DCEL) or equivalently half edge data structures (HDS) [PS85, dBvKOS97] when they discuss the implementation of plane maps. There are already standard implementations of *plane maps* like the CGAL HDS or embedded bidirected graphs in LEDA (similar design). We use an extended version of the CGAL HDS structure as the topological bottom layer of our Nef polyhedra. See the paper of

L. Kettner [Ket99] for an excellent review of different plane map representation and for the description of the adaptability of the CGAL HDS. To be more flexible we insert a decorator interface that homogeneously defines the functionality offered by the HDS. Geometrically we embed the vertices by means of extended points. Segments, rays, and lines are uniformly treated by the straight line embedding of edges incident to such vertices. We additionally add one face cycle of edges whose embedding corresponds to an infimaximal frame:

**Construction 1:** Consider a Nef polyhedron $P$ and the connected components of the faces of dimension zero to two. Assign plane map objects of corresponding dimension to each component and match the Nef incidence concept with the plane map incidence concept where possible. All objects corresponding to unbounded connected components of Nef edges and unbounded components of Nef 2-faces have incomplete incidence structures: edges miss end vertices and faces miss closed face cycles. To cure this, we add an infimaximal frame consisting of four uedges and four corner vertices. For all plane map edges $e$ that correspond to a Nef edge component extending to infinity along a ray $r$ we do the following: if $r$ is pruned by one of the corner vertices on the frame structure link $e$ to that vertex; otherwise $e$ obtains an additional terminating vertex in the relative interior of a uedge $e'$ that is part of the infimaximal frame where $r$ is meets the frame. $e'$ is split into two uedges by this vertex insertion. (For Nef edges that represent lines we do this at both ends). After all such edges are linked to the frame (respecting the embedding such that the adjacency list are order-preserving), all plane map faces corresponding to an unbounded component of a Nef face are cyclically bounded and their incidences structure can be completed. We call all edges and vertices that are part of the frame and the face outside of the frame that completes the subdivision combinatorially *infimaximal frame objects*.

To mark set membership, all objects of the plane map are *selectable*. All objects (vertices, edges, faces) obtain a marker labeling set inclusion or exclusion. The markers allow us to obtain the local pyramids associated to the plane map objects. The local view of a vertex is defined by its own marker, the markers of the edges in its adjacency list and markers of the faces in between these edges representing the neighborhood disc as explained above. The local view of an edge is defined by its mark and the two marks of its two incident faces. Finally the mark of a face maps to the trivial local view: the whole plane or the empty set depending on its selection flag. The selection markers of infimaximal frame objects have no geometric meaning and therefore those objects are always kept unselected.

As plane maps are implemented by bidirected graphs the incidence relation between edges and faces is encoded in an oriented fashion. Unoriented edges are implemented as pairs of oppositely directed halfedges where each such halfedge is incident to exactly one face. See the implementation description of plane maps in Section 4.6 for more information.

**Definition 9 (Data type):** A Nef polyhedron $P$ is stored as a selective plane map $(V,E,F)$ according to Construction 1. The objects of the plane map (vertices, edges, and faces) correspond to the connected components of the Nef faces of corresponding dimension and additionally to the infimaximal frame objects.

Each vertex $v \in V$ is embedded via an extended point $point(v)$. Each object $o \in V \cup E \cup F$ is contained in $P$ iff the selection mark $mark(o) == true$.

The feasibility of this definition can be seen as follows. Interpret $P$ as a set valued function $\phi$ on (open) half-spaces $H_1,\ldots,H_r$ that are combined by the operations $\cap$ and `cpl`. Let $h_i$ be the line bounding the half-space $H_i$. Now consider the arrangement build by the lines $\{h_i\}_i$ enclosed in a large enough frame. Interpret the arrangement $A(h_1,\ldots,h_r)$ inside the frame as a collection of relatively open convex cells of dimension 0 to 2. Consider any cell $c$. Any point of $c$ is either in $\phi(H_1,\ldots,H_r)$ or not. Mark all cells correspondingly. It should be clear that $A(h_1,\ldots,h_r)$ can be represented by a plane

map as described above. Now start a simplification process. Consider all edges $e$ (besides the ones on the frame box). If $e$ and the two faces incident to it have the same mark remove the edge and unify the faces (if not equal). Afterwards we iterate over all vertices and check if any vertex incident to two edges that are supported by the same line has the same local view as the points in the relative interior of the two incident edges. If this is the case, we unify the two edges and remove the vertex. Finally, we remove all vertices that are isolated and whose selection mark equals the one of the surrounding face. The final complex is just the data type described. When the simplification iteration terminates all points on vertices and edges have a local view that makes them low-dimensional faces of the Nef polyhedron.

The above algorithm is called *simplification* and is described in more detail when considering binary operations. There, we also give a runtime bound. Note that the local view properties of the vertices, edges, and faces of the plane map after the simplification process are a necessary condition of Definition 9.

**Lemma 4.4.1:** The representation of Definition 9 is unique.

*Proof.* Assume that there are two different plane maps $M(V,E,F)$ and $M'(V',E',F')$ representing the same Nef polyhedron $P$. If $P$ is the complete plane or the empty set then the plane maps consist of just one face inside the frame box and $F$ and $F'$ and their selection markers have to be equal. So assume otherwise. Then neither $P$ nor $\mathsf{cpl}P$ are empty. The boundary of $P$ and $\mathsf{cpl}P$ consists of vertices and edges. Assume that $M$ and $M'$ have a vertex at a point $x$ but the local views differ. Then, the represented point sets of $M$ and $M'$ differ and thus $P$ cannot be represented by both. If there is a point $x$ where $M$ has a vertex $v$ but $M'$ has not then obviously the local views are different too. Note that edges are terminated by vertices, and thus edges that are in $M$ but not in $M'$ already imply differences in the local view of their end vertices. The same holds when the edges are equal but the selection markers are not. Finally, note that due to the fact that the 1-skeleton of $M$ and $M'$ is equal, so are the faces (they are defined that way). Different selection markers on faces imply different local views in the vertices that are part of their closure. As a consequence $M$ and $M'$ have to be equal. □

Selective plane maps are the basic structure used to store Nef polyhedra. Remember that the edges and faces of a plane map are the connected components of the Nef edges and Nef 2-faces. For performance reasons we do not maintain the relationship between plane map objects and the corresponding abstract Nef faces which are defined as collections of the plane map objects with the same implicitly stored local pyramids. The *size* of a Nef polyhedron is the size of its underlying plane map (which is the number of vertices, edges, and faces).

For the binary operations we follow an approach as presented by Rossignac et al. [RO90]. In our case the approach is based on a generic plane sweep framework. A binary operation is basically split into three phases: subdivision – selection – simplification. The implementation is presented in the module *PM_overlayer*. An unary (topological) operation can be subdivided into two phases: selection – simplification.

We shortly present the abstract algorithmic ideas and the runtimes.

**subdivision** means for two plane maps $P_i(i = 0,1)$ to create a plane map $P$ with a minimal number of objects (vertices, edges, faces) such that each object of $P$ is supported by exactly one object of $P_i$ for $i = 0,1$. The subdivision is realized by a plane sweep of the objects of the 1-skeleta of $P_i$ followed by face creation and support determination. The time is dominated by the time for the sweep phase which is $O(n \log n)$ where $n$ is the size of the resulting subdivision.

**selection** with respect to the binary set operations means selecting the cells of the subdivision according to the logic of the underlying Boolean operation. With respect to unary topological set operations selection happens according to the logic of the topological unary operation. Selection is in both cases linear in $n$.

**simplification** means unification of subsets of cells that have the same local view. This phase has a quasi linear runtime due to the usage of a union-find data structure. Runtime is $O(n\alpha(kn,n))$ [6] for some small constant $k$.

**Theorem 4.4.1:** The result of a binary set operation (intersection, union, difference, symmetric difference) of two Nef polyhedra $P_0$ and $P_1$ can be calculated in time $O(n\log n)$ where n is the size of the overlay of $P_0$ and $P_1$. The result of an unary set operation (complement, boundary, interior, closure, regularization) can be constructed in time $O(n\alpha(kn,n))$ where $n$ is the size of the input structure.

The correctness and time bounds of our binary operations are based on three parts:

- The Sections 4.5.3 and 4.5.4 show the high-level composition of unary and binary set operations decomposed into phases.
- Section 4.7 provides the algorithmic modules for the subdivision, selection, and simplification phase. The correctness and resource argumentation is purely based on affine concepts and therefore our readers can trust their standard geometric intuition when verifying the correctness of those modules.
- Chapter 3 on the other hand shows that infimaximal frames allow us to use these algorithmic modules together with our extended objects.

The latter two observations together imply the correctness of the above theorem. The runtime lemmata of Section 4.7 imply the time bounds.

Now for our additional functionality like point location and ray shooting queries we need more than just the naked plane map structure described above. Looking at the literature for ray shooting there is the notion of segment walks which can be done easiest in convex subdivisions of the plane of bounded complexity [MMS94]. Our goal is to refine the basic plane map by such a structure. Note that this structure implies again faces, edges, and vertices, but of a much simpler fashion. However we do not want to lose the original character of the plane map. We might still be interested in the original face cycles. We store the refinement separate from the plane map. One solution to the segment walk problem is to use a constrained Delaunay triangulation of the edges and vertices of the HDS, and use any efficient point location structure for the location of the ray shooting start point.

## 4.5 Top Level Implementation

The whole implementation scheme is depicted in Figure 4.2. The main classes map to the abstract layers described above: geometry, plane maps, binary overlay, and point location. In the following we will mainly concentrate on the realization of the class *Nef_polyhedron_2* and the overlay modules *PM_overlayer* and *Segment_overlay_traits*. The functionality of *PM_decorator* is described by the concept in the appendix. The realization of the extended kernel is a topic of chapter 3. Point location is ommited here and can be studied in the full implementation report [See01].

[6] $\alpha(kn,n)$ is the extremly slow growing inverse of the Ackermann function used in the analysis of union-find data structures.

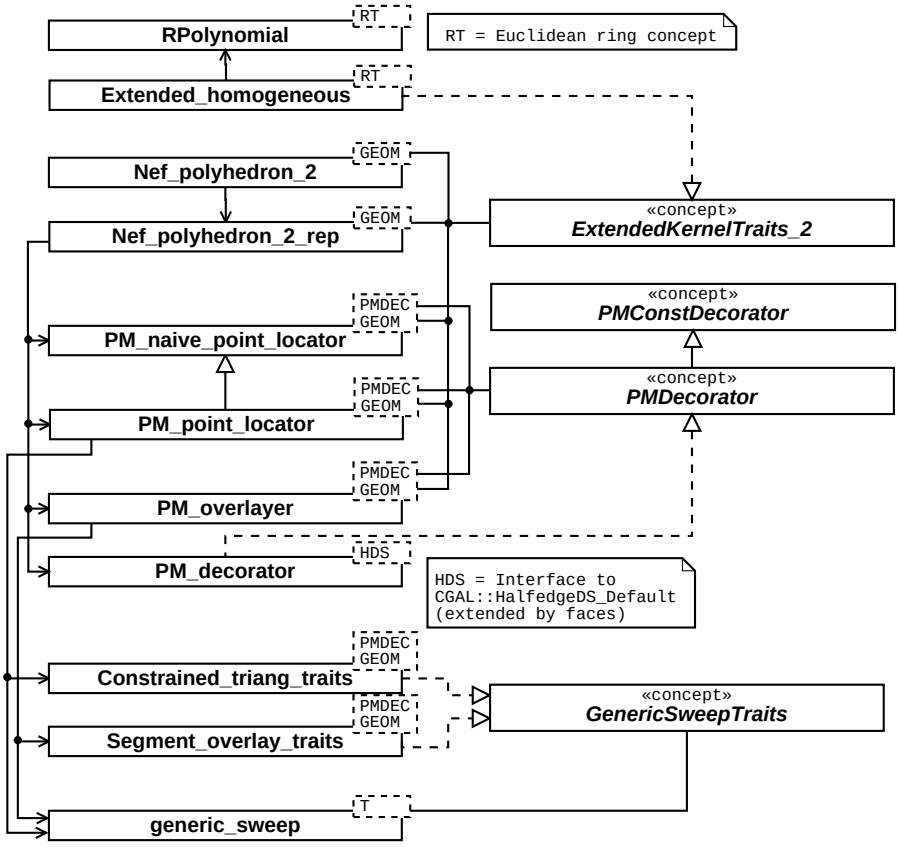



Figure 4.2: An UML diagram of the Nef polyhedron module. The main modules are the geometry *Extended_homogenous*<*RT*>, the plane map decorator *PM_decorator*<*HDS*>, the two algorithmic modules *PM_overlayer*<*PMDEC*, *GEOM*> and *PM_point_locator*<*PMDEC*, *GEOM*>.

### 4.5.1 The Polyhedron Class

Our data type *Nef_polyhedron_2<T>* is implemented as a smart pointer data type. Content such as a plane map object and a pointer to an optional point location object is stored in the class *Nef_polyhedron_2_rep<T>*. The traits template parameter *T* is specified by the concept *ExtendedKernelTraits_2* as presented on page 187.

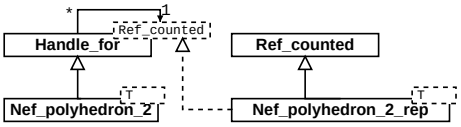



Figure 4.3: The smart pointer realization of data type *Nef_polyhedron_2*.

Within the scope of *Nef_polyhedron_2_rep<T>* all auxiliary classes are instantiated. The plane map type is based on the CGAL HDS and uses two traits classes *HDS_traits* and *HDS_items*. The former carries attributes the latter carries the (fixed) models for vertices, edges, and faces. Compare the extensions or their standard design in Section 4.6.

⟨*nef rep types*⟩≡
```
struct HDS_traits {
  typedef typename T::Point_2 Point;
  typedef bool                Mark;
};
typedef CGAL_HALFEDGEDS_DEFAULT<HDS_traits,HDS_items> Plane_map;
typedef CGAL::PM_const_decorator<Plane_map>           Const_decorator;
typedef CGAL::PM_decorator<Plane_map>                 Decorator;
typedef CGAL::PM_naive_point_locator<Decorator,T>     Slocator;
typedef CGAL::PM_point_locator<Decorator,T>           Locator;
typedef CGAL::PM_overlayer<Decorator,T>               Overlayer;
```

In the class *Nef_polyhedron_2<T>* all geometric types are obtained from the geometric traits class *T*. *T* contains affine types that are part of the interface but also the extended types that are used in the infimaximal framework. The *Standard*-prefixed types from within *T* become the interface types of *Nef_polyhedron_2<T>*. The non-prefixed[7] types within *T* become the extended types within *Nef_polyhedron_2<T>*.

⟨*nef interface types*⟩≡
```
typedef Nef_polyhedron_2<T> Self;
typedef Handle_for< Nef_polyhedron_2_rep<T> > Base;
typedef typename T::Point_2   Extended_point;
typedef typename T::Segment_2 Extended_segment;
typedef typename T::Standard_line_2 Line;
typedef typename T::Standard_point_2 Point;
typedef typename T::Standard_direction_2 Direction;
typedef typename T::Standard_aff_transformation_2  Aff_transformation;
```

[7] T is used as a traits class in our generic geometry based modules like *PM_overlayer<T>*. Therefore, the extended types conform to a simpler naming scheme within *T*.

We import the type *Plane_map* and all decorator types like *Decorator*, *Overlayer*, *Locator*, *Slocator* from the representation type *Nef_polyhedron_2_rep*. Additionally, we import handles and iterators from *Decorator* (we do not list those typedef statements).

### 4.5.2 Creating Polyhedra

We provide the construction methods for basic polyhedra. These are the empty set, the whole plane, open and closed half-planes, and construction of simple polygonal chains (Jordan Curves) where the modeled point set can be the bounded or unbounded part for the plane and the set can be open or closed.

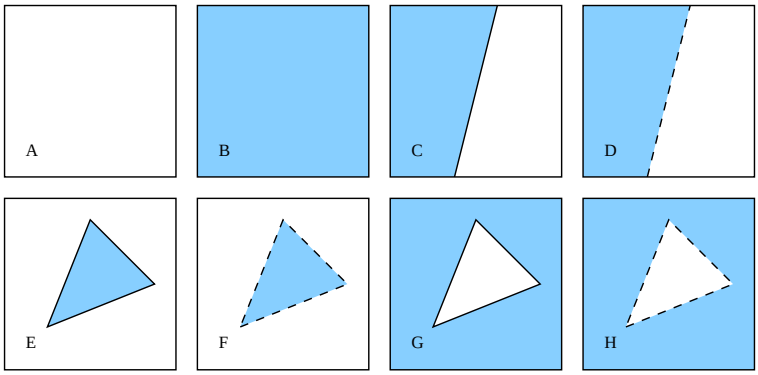



Figure 4.4: Elementary Nef polyhedra: (A) the empty set, (B) the whole plane, (C/D) a closed/open half-plane, (E/F) a closed/open bounded polygon, (G/H) the plane with a closed/open polygonal hole.

The construction of simple Nef polyhedra is reduced to the overlay of a list of extended segments, the creation of the 2-faces, followed by setting the attribute marks that code set inclusion. The first task is implemented in our overlayer module *PM_overlayer*<>::*create*$(s,e,DA)$ where $S = tuple[s,e)$ is the set of segments and *DA* is a data accessor that allows us to link plane map edges to the segments in *S*.

Finally, we only have to take care of the correct marks of the plane map objects with respect to the construction information from our constructor interface. Note that by using the overlayer module we obtain all output properties of the plane map created from that module.

All Nef polyhedra obtain an infimaximal frame embedded by four extended segments. We encapsulate this into the following operation. See the extended geometry module for the definition of this frame.

⟨*nef protected members*⟩+≡

```
typedef std::list<Extended_segment>      ES_list;
typedef typename ES_list::const_iterator ES_iterator;
void fill_with_frame_segs(ES_list& L) const
{ L.push_back(Extended_segment(EK.SW(),EK.NW()));
  L.push_back(Extended_segment(EK.SW(),EK.SE()));
  L.push_back(Extended_segment(EK.NW(),EK.NE()));
  L.push_back(Extended_segment(EK.SE(),EK.NE()));
}
```

We want to establish a link between a particular extended segment and its corresponding edge in the plane map. Our overlay module allows us to get a grip on this relation by means of a data accessor that is passed to the overlay algorithm. The class *Link_to_iterator* is a model for that data accessor concept. An object *D* of this type can store an iterator referencing a segment. Passed to the *PM_overlayer*<>::*create*(...) method it stores the corresponding edge after the executed overlay as its member *D._e*. *Link_to_iterator* also initializes all marks of the newly created skeleton objects.

We do not show the creation of the empty set or the full plane. This is just a trivial case of the following half-plane construction. We come to the construction of a half-plane. A user describes an open or closed half-plane by an oriented line *l*. To create a plane map representing it we overlay a frame box plus an extended segment splitting the box into two faces along *l*. The overlayer module *PM_overlayer*<>::*create*(...) creates the plane map (including faces) out of the list of extended segments passed to it where no object is selected (concerning membership). The data accessor *Link_to_iterator I* obtains the edge *I._e* of *pm*( ) corresponding to −−*L.end*( ) (the iterator pointing to the extended segment which is the line) during the *create* phase. We can use that edge to mark its adjacent face and the edge itself according to the *line* flag.

⟨*nef interface operations*⟩+≡

```
Nef_polyhedron_2(const Line& l, Boundary line = INCLUDED) : Base(Nef_rep())
{
  ES_list L;
  fill_with_frame_segs(L);
  Extended_point ep1 = EK.construct_opposite_point(l);
  Extended_point ep2 = EK.construct_point(l);
  L.push_back(EK.construct_segment(ep1,ep2));
  Overlayer D(pm());
  Link_to_iterator I(D, --L.end(), false);
  D.create(L.begin(),L.end(),I);
  Halfedge_handle el = I._e;
  if ( D.point(D.target(el)) != EK.target(L.back()) )
    el = D.twin(el);
  D.mark(D.face(el)) = true;
  D.mark(el) = bool(line);
}
```

The construction of a simple polygon defined by an iterator range of standard affine points (the value type of *Forward_iterator* is *Point*) follows the same idea, however we also accept degenerate polygons.

The iterator range of points can confront us with the following cases: (1) the list is empty, (2) the list has only one point, (3) the list contains at least two points spanning line segments where the following cases and problems can occur: (a) the segment(s) has (have) affine dimension 1 (the hull is a segment). (b) the segments enclose a simple polygon. (c) the segments enclose no simple polygon (touching or intersecting inside). We cover one point, two points spanning a segment and *n* points spanning a simple polygon (which we do not check). The construction of the plane map still succeeds when *P* is not simple as the overlayer module just constructs the planar subdivision implied by the segments in the iterator range. However, the face marks will in general be incorrect.

⟨*nef interface operations*⟩+≡

```
template <class Forward_iterator>
Nef_polyhedron_2(Forward_iterator it, Forward_iterator end,
  Boundary b = INCLUDED) : Base(Nef_rep())
```

```
{
  ES_list L;
  fill_with_frame_segs(L);
  bool empty = false;
  if (it != end)
    ⟨fi ll segment list L⟩
  else empty = true;
  Overlayer D(pm());
  Link_to_iterator I(D, --L.end(), true);
  D.create(L.begin(),L.end(),I);
  ⟨mark face and boundary⟩
}
```

We fill $L$ with the segments cyclically spanned by the points in the input iterator range.

⟨*fi ll segment list L*⟩≡

```
{
  Extended_point ef, ep = ef = EK.construct_point(*it);
  Forward_iterator itl=it; ++itl;
  if (itl == end) // case only one point
    L.push_back(EK.construct_segment(ep,ep));
  else { // at least one segment
    while( itl != end ) {
      Extended_point en = EK.construct_point(*itl);
      L.push_back(EK.construct_segment(ep,en));
      ep = en; ++itl;
    }
    L.push_back(EK.construct_segment(ep,ef));
  }
}
```

We create the marks via the object stored in the *Link_to_iterator* object $I$. The object is determined by the last segment $s$ in $L$. If that segment is trivial then $I.\_v$ stores the corresponding vertex. If $s$ is non-trivial then $I.\_e$ contains the edge supported by the segment. We only have to extract the correct halfedge of the edge twins. Then, we can mark the face and the boundary accordingly.

⟨*mark face and boundary*⟩≡

```
if ( empty ) {
  D.mark(++D.faces_begin()) = !bool(b); return; }
if ( EK.is_degenerate(L.back()) ) {
  D.mark(D.face(I._v)) = !bool(b); D.mark(I._v) = b;
} else {
  Halfedge_handle el = I._e;
  if ( D.point(D.target(el)) != EK.target(L.back()) )
    el = D.twin(el);
  D.set_marks_in_face_cycle(el,bool(b));
  if ( D.number_of_faces() > 2 ) D.mark(D.face(el)) = true;
  else                           D.mark(D.face(el)) = !bool(b);
}
clear_outer_face_cycle_marks();
```

We leave out the simple implementation of copy construction, cloning and basic operations like *clear*( ), *is_empty*( ), and *is_plane*( ).

### 4.5.3 Unary Operations

For the unary operations on Nef polyhedra we implement several class-modifying methods. They can be chained together to larger units. We thereby save cloning operations if the representation object is only referenced by one handle. Note that the first line of any modifying operation has to check if the representation object is shared by several handles. If the representation object is shared, the plane map has to be cloned before modification. We first implement the three operations `cpl`, `int`, and `bd`.

As our planar Nef polyhedra completely partition the plane, the complement operation is easy to implement by an inversion (Boolean flip) of the selection markers. Note that this conforms to the result [Nef78, theorem 6;14] in which Nef showed that the low-dimensional faces of $P$ and $\mathtt{cpl}P$ (in their common boundary) are the same and the full-dimensional faces that are part of $P$ or $\mathtt{cpl}P$ obviously exchange their role (if non-empty). The local pyramid in any point $x$ of the plane is inverted by the Boolean flips according to [Nef78, theorem 3;15] $(\mathtt{cpl}P)^x = \mathtt{cpl}P^x$. Due to the special role[8] of the outer face and the edges and vertices that are part of the frame box we keep all objects of the frame unmarked (operation *clear_outer_face_cycle_marks*). Note that just by flipping we do not spoil the local views properties as specified in Lemma 4.3.1. Thus, we do not have to simplify here.

⟨*nef interface operations*⟩+≡

```
void extract_complement()
{
  if ( ptr->is_shared() ) clone_rep();
  Overlayer D(pm());
  Vertex_iterator v, vend = D.vertices_end();
  for(v = D.vertices_begin(); v != vend; ++v)      D.mark(v) = !D.mark(v);
  Halfedge_iterator e, eend = D.halfedges_end();
  for(e = D.halfedges_begin(); e != eend; ++(++e)) D.mark(e) = !D.mark(e);
  Face_iterator f, fend = D.faces_end();
  for(f = D.faces_begin(); f != fend; ++f)         D.mark(f) = !D.mark(f);
  clear_outer_face_cycle_marks();
}
```

The interior $\mathtt{int}P$ of a point set $P$ is the set of all points where an open ball of infinitesimal radius (a neighborhood) is contained in the point set. This is not the case for all low-dimensional faces of the plane map. Accordingly (see also [Nef78, theorem 3;19]), we have to keep all selected full-dimensional faces of $P$ and all objects of the 1-skeleton have to be deselected. Afterwards the simplification operation minimizes the structure and makes it again consistent with Definition 9. For example, marked isolated vertices within non-marked faces and marked edges within such faces are first unmarked and then deleted in the *simplify*( ) operation. The simplification operation has to exempt edges of the infimaximal frame. An object *Except_frame_box_edges*$(P)$ has a function operator method $bool\ operator(\,)(Halfedge\_handle\ e)$ that returns true iff the edge $e$ of the plane map $P$ is part of the frame box. Thereby within *simplify* a removal of edges is only executed on edges that partition the interior of the frame box.

[8] No affine object can be placed on or outside the infimaximal frame.

⟨*nef interface operations*⟩+≡
```
void extract_interior()
{
  if ( ptr->is_shared() ) clone_rep();
  Overlayer D(pm());
  Vertex_iterator v, vend = D.vertices_end();
  for(v = D.vertices_begin(); v != vend; ++v)      D.mark(v) = false;
  Halfedge_iterator e, eend = D.halfedges_end();
  for(e = D.halfedges_begin(); e != eend; ++(++e)) D.mark(e) = false;
  D.simplify(Except_frame_box_edges(pm()));
}
```

The boundary of a point set $P$ is defined to be the intersection $\mathsf{clos}\,P \cap \mathsf{clos}\,\mathsf{cpl}\,P$. This is the set of all points that have a nonempty neighborhood with $\mathsf{int}\,P$ or $\mathsf{ext}\,P$. Any point $x$ on the 1-skeleton of the plane map has this property due to the properties of its local pyramid $P^x$. The boundary $\mathsf{bd}\,P$ is thus obtained by selecting all low-dimensional objects and then deselecting all 2-faces. Finally we cope with the frame and simplify the structure.

⟨*nef interface operations*⟩+≡
```
void extract_boundary()
{
  if ( ptr->is_shared() ) clone_rep();
  Overlayer D(pm());
  Vertex_iterator v, vend = D.vertices_end();
  for(v = D.vertices_begin(); v != vend; ++v)      D.mark(v) = true;
  Halfedge_iterator e, eend = D.halfedges_end();
  for(e = D.halfedges_begin(); e != eend; ++(++e)) D.mark(e) = true;
  Face_iterator f, fend = D.faces_end();
  for(f = D.faces_begin(); f != fend; ++f)         D.mark(f) = false;
  clear_outer_face_cycle_marks();
  D.simplify(Except_frame_box_edges(pm()));
}
```

Finally, we use the above operations for *closure* and *regularization*. The closure of $P$ can be reduced to the operations *interior* and *complement* as $\mathsf{clos}\,P = \mathsf{cpl}\,\mathsf{int}\,\mathsf{cpl}\,P$. The regularization of $P$ is defined as $\mathsf{clos}\,\mathsf{int}\,P$.

⟨*nef interface operations*⟩+≡
```
void extract_closure()
{
  extract_complement();
  extract_interior();
  extract_complement();
}
void extract_regularization()
{
  extract_interior();
  extract_closure();
}
```

The constructive interface methods are just mapped to the corresponding extract methods.

⟨*nef interface operations*⟩+≡
```
Nef_polyhedron_2<T> complement() const
{ Nef_polyhedron_2<T> res = *this;
  res.extract_complement();
  return res;
}
```

All other operations like *interior*( ), *closure*( ), *boundary*( ), and *regularization*( ) are implemented accordingly.

### 4.5.4 Binary Set Operations

We follow Rossignac and O'Connor [RO90] and split the binary operations into three phases subdivision – selection – simplification. For the implementation see the module *PM_overlayer*<> in Section 4.7. The subdivision phase creates the overlay of the two input structures. This overlay has the property that each object (vertex, edge, face) has exactly one object from each input structure that supports it. This proposition is proven in Lemma 4.7.1. After the subdivison each object of the resulting plane map knows its mark in each of the two input structures and can thereby be qualified with respect to each input structure. The binary set operation is then reduced to a Boolean predicate on these marks. The resulting structure can be a planar partition that is not a legal Nef polyhedron due to the fact that plane map boundary objects can have local views that contradict Lemma 4.3.1. Violations are fixed in the simplification phase without changing the represented point set. This simplification makes the plane map representation minimal with respect to the number of its objects and again consistent with Definition 9.

The above scheme refers to the theory as presented by Nef who showed the following general lemma.

**Lemma 4.5.1:** Let $P_0$, $P_1$ be polyhedra. Then every face of $P = P_0 \cap P_1$ is the union of intersections of faces of $P_0$ and $P_1$.

The proof follows [Nef78, Satz 6;16]. As a consequence the simplification just unions objects within $P$ to form the connected components of Nef faces.

To implement the binary set operations we use functors[9] that carry the underlying Boolean logic.

⟨*boolean classes*⟩≡
```
struct AND { bool operator()(bool b1, bool b2)  const { return b1&&b2; }  };
struct OR { bool operator()(bool b1, bool b2)   const { return b1||b2; }  };
struct DIFF { bool operator()(bool b1, bool b2) const { return b1&&!b2; } };
struct XOR { bool operator()(bool b1, bool b2)  const
                                          { return (b1&&!b2)||(!b1&&b2); } };
```

⟨*nef interface operations*⟩+≡
```
Nef_polyhedron_2<T> intersection(const Nef_polyhedron_2<T>& N1) const
{ Nef_polyhedron_2<T> res(pm(),false); // empty, no frame
  Overlayer D(res.pm());
  D.subdivide(pm(),N1.pm());
```

[9] a short form for function objects.

```
    AND _and; D.select(_and);
    res.clear_outer_face_cycle_marks();
    D.simplify(Except_frame_box_edges(res.pm()));
    return res;
  }
```

Join, difference, and symmetric difference follow similar schemes based on *OR*, *DIFF*, and *XOR*. We do not show the implementation of the operators $*, +, -, \hat{}, !$ which map to the above operations.

### 4.5.5 Binary Comparison Operations

All set comparison operations are reduced to binary operations followed by an empty-set test. For two Nef polyhedra $P_1$, $P_2$ it holds

$$\begin{aligned} P_1 = P_2 &\Leftrightarrow \mathit{symmetric\_difference}(P_1, P_2) = \emptyset \\ P_1 \subseteq P_2 &\Leftrightarrow \mathit{difference}(P_1, P_2) = \emptyset \\ P_1 \subsetneq P_2 &\Leftrightarrow \mathit{difference}(P_1, P_2) = \emptyset \ \wedge \ \mathit{difference}(P_2, P_1) \neq \emptyset \end{aligned}$$

In our specification $\subseteq$ is *operator* $\leq$, and $\subsetneq$ is *operator* $<$. The other operations are symmetric.

⟨*nef interface operations*⟩+≡

```
bool operator==(const Nef_polyhedron_2<T>& N1) const
{ return symmetric_difference(N1).is_empty(); }
bool operator!=(const Nef_polyhedron_2<T>& N1) const
{ return !operator==(N1); }
bool operator<=(const Nef_polyhedron_2<T>& N1) const
{ return difference(N1).is_empty(); }
bool operator<(const Nef_polyhedron_2<T>& N1) const
{ return difference(N1).is_empty() && !N1.difference(*this).is_empty(); }
```

### 4.5.6 Point location and Ray shooting

Let $P$ be the plane map underlying our Nef polyhedron stored in the $*$*this* object. The result of a point location query with an affine point $p$ is the object of $P$ whose embedding contains $p$. Ray shooting queries come in two flavors. One variant starts the ray shot in a point $p$ and determines the closest object of $P$ in direction $d$ that is in the set (determined by the selection mark). The other variant determines the closest 1-skeleton object in direction $d$. The point location and ray shooting functionality is taken from the two point location classes *PM_point_locator*<> and *PM_naive_point_locator*<>. All operations can choose between the two approaches by a mode flag $m$. The default is location as implemented by *PM_point_locator*<>. That class uses a further subdivision of $P$ by a locally minimized weight constrained triangulations (LMWT) to allow so-called segment walks. The LMWT is calculated on demand, when the first point location or ray shooting operation is called. The naive point location method is based on a global examination of all objects of $P$ to find the one that contains

*p*. The type *Object_handle* is a polymorphic handle that can reference vertices, edges, and faces. Conversion is done by an assign operation similarly to the polymorphic CGAL type *Object*.

⟨*nef interface operations*⟩+≡

```
bool contains(Object_handle h) const
{ Slocator PL(pm()); return PL.mark(h); }
bool contained_in_boundary(Object_handle h) const
{ Vertex_const_handle v;
  Halfedge_const_handle e;
  return  ( CGAL::assign(v,h) || CGAL::assign(e,h) );
}
```

The chosen point location method depends on *m*. In the non-naive case the locator object is initialized by a call to *init_locator*( ). The corresponding locator object is stored in the representation object for further usage in iterated queries and can be accessed by the *locator*( ) method. The naive locate operation requires a segment as input that intersects the 1-skeleton of the plane map.

⟨*nef interface operations*⟩+≡

```
Object_handle locate(const Point& p, Location_mode m = DEFAULT) const
{
  if (m == DEFAULT || m == LMWT) {
    ptr->init_locator();
    Extended_point ep = EK.construct_point(p);
    return locator().locate(ep);
  } else if (m == NAIVE) {
    Slocator PL(pm(),EK);
    Extended_segment s(EK.construct_point(p),
                       PL.point(PL.vertices_begin()));
    return PL.locate(s);
  }
  CGAL_assertion_msg(0,"location mode not implemented.");
  return Object_handle();
}
```

The ray shooting operation determines the closest object of *P* that is marked and hit by a ray shot from the point *p* in direction *d*. The search is delegated to the corresponding member of the locator object. The class *INSET* is the predicate class that stops the ray shot when a marked object is hit. See the manual page of the locator classes for its concept.

⟨*nef interface operations*⟩+≡

```
struct INSET {
  const Const_decorator& D;
  INSET(const Const_decorator& Di) : D(Di) {}
  bool operator()(Vertex_const_handle v) const { return D.mark(v); }
  bool operator()(Halfedge_const_handle e) const { return D.mark(e); }
  bool operator()(Face_const_handle f) const { return D.mark(f); }
};
Object_handle ray_shoot(const Point& p, const Direction& d,
                        Location_mode m = DEFAULT) const
{
```

```
  if (m == DEFAULT || m == LMWT) {
    ptr->init_locator();
    Extended_point ep = EK.construct_point(p),
                   eq = EK.construct_point(p,d);
    return locator().ray_shoot(EK.construct_segment(ep,eq),
                               INSET(locator()));
  } else if (m == NAIVE) {
    Slocator PL(pm(),EK);
    Extended_point ep = EK.construct_point(p),
                   eq = EK.construct_point(p,d);
    return PL.ray_shoot(EK.construct_segment(ep,eq),INSET(PL));
  }
  CGAL_assertion_msg(0,"location mode not implemented.");
  return Object_handle();
}
```

A similar implementation is used for *ray_shoot_to_boundary*. Note that we only use a different predicate *INSKEL*.

⟨*nef interface operations*⟩+≡

```
struct INSKEL {
  bool operator()(Vertex_const_handle) const { return true; }
  bool operator()(Halfedge_const_handle) const { return true; }
  bool operator()(Face_const_handle) const { return false; }
};
Object_handle ray_shoot_to_boundary(const Point& p, const Direction& d,
              Location_mode m = DEFAULT) const
{
  if (m == DEFAULT || m == LMWT) {
    ptr->init_locator();
    Extended_point ep = EK.construct_point(p),
                   eq = EK.construct_point(p,d);
    return locator().ray_shoot(EK.construct_segment(ep,eq),INSKEL());
  } else if (m == NAIVE) {
    Slocator PL(pm(),EK);
    Extended_point ep = EK.construct_point(p),
                   eq = EK.construct_point(p,d);
    return PL.ray_shoot(EK.construct_segment(ep,eq),INSKEL());
  }
  CGAL_assertion_msg(0,"location mode not implemented.");
  return Object_handle();
}
```

To examine the plane map underlying the Nef polyhedron the user can obtain a decorator object that has read-only access to *pm*( ). Thus, modifications can only take place via the interface operations of *Nef_polydron_2*.

⟨*nef interface operations*⟩+≡

```
Explorer explorer() const { return Explorer(pm(),EK); }
```

### 4.5.7 Visualization

At last we provide a drawing routine for Nef polyhedra in a LEDA window. We want to draw faces, edges, vertices in this order, where faces are maximally connected point sets bounded by one outer face cycle and maybe by several inner hole cycles. We draw objects which are in our point set black and objects which are not in our pointset by a light color. Note that we face the following problem. Our window represents a rectangular view to our square frame, which is large enough to make the topology on this boundary constant. Imagine making our frame big enough and then shrinking it slowly down to zero. For each ray with slope not equal to one and not containing the origin there is a value $R$ when the ray tip on the frame leaves its correct frame segment. If we want to prevent topological difficulties when drawing the polyhedron we have to keep our frame radius above the minimum $R_m$. Thus visualization determines this $R_m$ and sets the internal evaluation parameter to this value. Then all points on the frame and segments containing such points have fixed coordinates and can be drawn. For the concrete technical details of drawing the objects see the class *PM_visualizor*<>.

⟨*window stream output*⟩≡

```
static long frame_default = 100;
static bool show_triangulation = false;

template <typename T>
CGAL::Window_stream& operator<<(CGAL::Window_stream& ws,
const Nef_polyhedron_2<T>& P)
{
  typedef Nef_polyhedron_2<T> Polyhedron;
  typedef typename T::RT RT;
  typedef typename T::Standard_RT Standard_RT;
  typedef typename Polyhedron::Topological_explorer TExplorer;
  typedef typename Polyhedron::Point            Point;
  typedef typename Polyhedron::Line             Line;
  typedef CGAL::PM_BooleColor<TExplorer> BooleColor;
  typedef CGAL::PM_visualizor<TExplorer,T,BooleColor> Visualizor;

  TExplorer D = P.explorer();
  const T& E = Nef_polyhedron_2<T>::EK;

  Standard_RT frame_radius = frame_default;
  E.determine_frame_radius(D.points_begin(),D.points_end(),frame_radius);
  RT::set_R(frame_radius);
  Visualizor PMV(ws,D); PMV.draw_map();
  ⟨draw the refining constrained triangulation⟩
  return ws;
}
```

Drawing of the constrained triangulation is done depending on the static variable *show_triangulation*. Of course such animation requires the necessary preprocessing with the locator object.

⟨*draw the refining constrained triangulation*⟩≡
```
if (show_triangulation) {
  P.init_locator();
  Visualizor V(ws,P.locator().triangulation());
  V.draw_skeleton(CGAL::BLUE);
}
```

### 4.5.8 Input and Output

Standard input and output is done by the plane map I/O class *PM_io_parser*. See Section 4.6 for more information.

### 4.5.9 Hiding extended geometry

The plane map explorer provides an interface that masks the properties of our extended kernel by reintroducing purely affine objects. We want to provide a simple interface for users who are not interested in the detailed features of extended objects. The methods of the class *PM_explorer*<> allow queries to the category of vertices and edges such as "Is a vertex embedded in the affine space or on the frame box?" or "Is an edge part of the affine structure or part of the frame box". Its implementation trivially maps to the operations of the extended kernel. See its interface in Section 4.2.2 on page 178.

## 4.6 Plane Map Implementation

We present some implementation details of our plane map decorator that provides an abstract interface to a plane map. The abstract interface of our plane map data type is sufficiently specified in the manual page. We sketch how we implement this interface by the CGAL halfedge data structure. We use the new HDS design as described in the design paper [Ket99]. The paper contains also a survey of classical plane map implementations and a motivation for the HDS design. The generic HDS collection allows to choose different flavors of HDS structures. A user can specify if she uses explicit vertex or face objects and how the iteration facilities are implemented. For the topological Nef layer we choose the default implementation including vertex and face objects, however the offered design is limited to one single face cycle bounding a face. Our definition of plane maps requires to have multiple face cycles and also trivial face cycles in form of isolated vertices. We do not want to bore the reader with the technical details of the implementation but we describe the extension process from the functionality of the default HDS design to our plane map data type. Fortunately, the CGAL HDS allows a user to extend the functionality by extending the objects (vertices, halfedges, faces). The types are transported into the container type HDS in a so-called items class; in our case called *HDS_items*. The possibility of this extension is one advantage of the generic design.

Figure 4.5 presents the default layout of the three objects. The interface methods map to member variables. A vertex $v$ stores an incident edge $e$ such that $v.halfedge() = e$ and $e.vertex() = v$. Dually symmetrical a face $f$ stores an edge $e$ in its bounding face cycle: $f.halfedge() = e$ and $e.face() = f$. The additional links of an edge $e$ create the topological structure of the graph. $e.opposite()$ is used to make the graph bidirected and $e.next()$ and $e.prev()$ are used for the circular ordering of edges in the face cycle of a face.

The extended structure in Figure 4.6 adds the possibility to assign multiple face cycles as a boundary to a face $f$ to the above structure. We give generic container access by means of two iterator ranges.

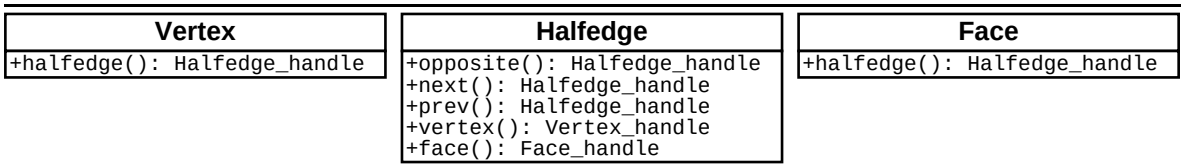



Figure 4.5: The default HDS design

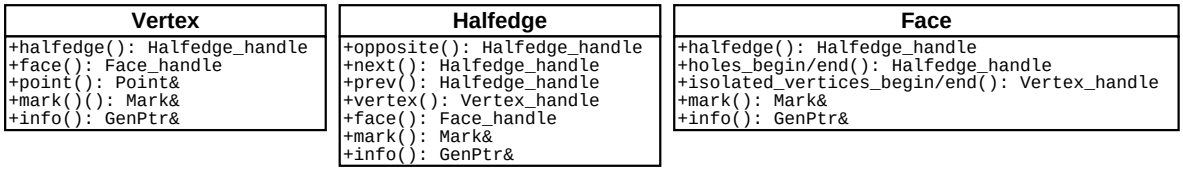



Figure 4.6: The extended HDS design

The range $[\ f.holes\_begin(\ ),\ f.holes\_end(\ )\ )$ stores halfedges that can be used as entry points into disjoint face cycles. Accordingly, the range $[\ f.isolated\_vertices\_begin(\ ),\ f.isolated\_vertices\_end(\ )\ )$ maintains a set of vertices in the interior of $f$. Such vertices $v$ also store their containing face by $v.face(\ ) = f$. Note that our implementation requires that insertion or deletion operations in the two sets are constant time operations. We do not show the details. Note that all three objects are attributed twice in addition to the topological linkage. All objects store a mark and a generic slot of type $GenPtr = void*$ for further data association. This two extensions are mandatory for the Nef structure. The $mark(\ )$ slots map to set inclusion flags. The $info(\ )$ slots allow us to associate temporary information required for the binary overlay of two structures. Finally, vertices carry a point representing their embedding into the plane.

## Decorator classes

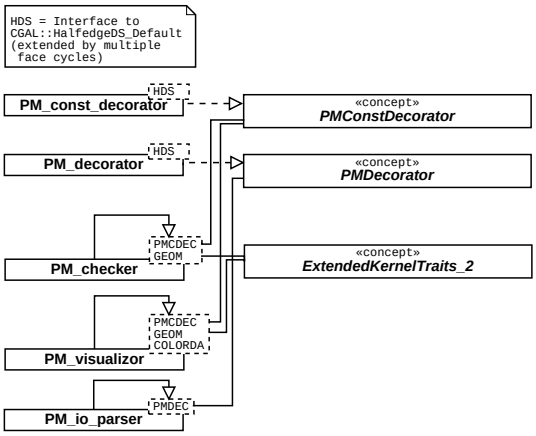



Figure 4.7: The decorator family defi ning the interface to the HDS data structure

We interface the HDS via decorator classes that encapsulate a certain functionality. The classes are depicted in Figure 4.7. We implemented the two main concepts *PMConstDecorator* and *PMDecorator* on top of the CGAL HDS. The first gives read-only access to the HDS the second provides manipulation operations. The concepts carry their interface into the additional modules *PM_checker*, *PM_io_parser*, *PM_visualizor*. Whenever geometric kernel operations are needed in the module as for example in the checker, we add a template parameter carrying geometric kernel methods. The *COLORDA* template parameter in the visualizor module *PM_visualizor* allows the adaptation of the drawing of plane maps. We elaborate on some details of the modules but do not show the whole implementation.

### PM_const_decorator - the read-only interface

*PM_const_decorator*<> realizes non-mutable access to plane maps. It provides interface operations on the objects as presented in our concepts section on page 179. The sole link to geometry is the embedding via a point type. All circular structures are realized via circulators (the variation of iterators as introduced in CGAL). The only method that carries more involved coding is the integrity check operation. That operation checks the sanity of the link structure coding the incidence relations of vertices, edges, and faces and additionally checks the topological planarity of the structure by checking that the genus of the plane map is zero. The integrity check of the topological decorator does the following:

- all vertices are partitioned into two sets by the *is_isolated*( ) predicate. All isolated vertices $v$ have face links where $v$ is in the isolated vertices list of $v \rightarrow face(\,)$. All non-isolated vertices are bound to adjacency lists by their halfedge link.
- for all vertices $v$ we check that $source(A(v)) == v$
- for all edges $e$ we check that $twin(twin(e)) == e$
- we check that the Euler formula is correctly fulfilled. Let $n_v$ be the number of vertices, $n_e$ be number of edges (= number of halfedges divided by 2), $n_f$ be the number of faces, $n_{fc}$ be the number of face cycles, and $n_{cc}$ be the number of connected components of the map. Then at first we have $n_f = n_{fc} - n_{cc} + 1$ and we check that $n_v - n_e + n_f = 1 + n_{cc}$. Note that we have to cope with isolated vertices. They are counted in our connected component number $n_{cc}$ and in $n_v$.

See Chapter 8 of the LEDA book for an elaborate treatment of this check.

### PM_checker - checking geometric properties

Our checker mainly realizes the integrity checks of the basic properties of the plane map like that of an order-preserving embedding or the forward-prefix property of the adjacency lists. We also added a checker method that examines if a plane map represents a triangulation of its vertices. The implemented methods are

```
void check_order_preserving_embedding(Vertex_const_handle v) const;
void check_forward_prefix_condition(Vertex_const_handle v) const;
void check_order_preserving_embedding() const;
void check_is_triangulation() const;
```

The methods check the basic properties that we require from a plane map. The task to check if our plane map actually is a triangulation of its vertices follows the ideas as presented in [MNS+99] and the LEDA book.

### PM_decorator - manipulating the plane map

*PM_decorator*<> gives mutable access to a plane map. Apart from standard operations the interface also provides operations that are very specially designed for the updates needed in our sweep framework or in the simplification phase of our binary operations. Some operations allow changing the incidence of plane map objects only partially e.g. create an edge that is only linked to a vertex at its source. With these operations one has to be careful not to spoil the plane map structure. The advantage is that we do not need superflous allocations of objects that are only needed temporarily.

The implementation of most of the operations is straight-forward. The only operation that should be mentioned is the clone operation for plane maps. As the generic HDS container does not know the layout of the objects that it maintains, a copy construction is hard to realize for the general case. In the rare case where we actually need to copy a plane map, we use the methods:

```
void clone(const HDS& H);
template <typename LINKDA>
void clone_skeleton(const HDS& H, const LINKDA& L)
```

Both methods basically work in two stages. Let $H'$ be the target copy of $H$. First each object $o$ in $H$ is cloned into an object $o'$ in $H'$ whose links still point to objects in $H$. We store the correspondance of $o$ to $o'$ in a map $M(o) = o'$. Then in the second stage we iterate all objects $o'$ in $H'$ and replace the links to the objects in $H$ by the corresponding objects in $H'$ via the map. The result is an isomorphic structure. Note that due to the fact that the *prev-next* links of the halfedges also code the embedding, this isomorphy is also topological and not only combinatorial [Die97]. Of course the geometric embedding of the vertices and the attributed marks are just transferred. The second cloning operation just extracts an topological isomorphic 1-skeleton from a full-fledged plane map. In that operation we also provide access to the newly created objects by an additional data accessor $L$. The *LINKDA* concept requires the methods:

```
struct LINKDA {
  void operator()(Vertex_handle vn, Vertex_const_handle vo) const;
  void operator()(Halfedge_handle hn, Halfedge_const_handle ho) const;
}
```

where *vn*, *hn* are the cloned objects in $H'$ and *vo*, *ho* are the original objects of $H$. $L$ can now be used to get a hand on the cloning process on the object level. The method is used to obtain an isomorphic graph structure that can be used for further subdivision (e.g. point location in constrained triangulations). We leave out the details, as its design is mainly determined by the design of the CGAL HDS.

### PM_io_parser - stream input and output

The input and output is mainly triggered by a decorator which has the control over the I/O format and does some basic parsing when reading input. The class template *PM_io_parser*<*PMDEC*> has two constructors and two corresponding actions on the streams obtained on construction:

```
PM_io_parser(std::istream& is, Plane_map& H);
void read();
PM_io_parser(std::ostream& os, const Plane_map& H);
void print() const;
```

The template parameter refers to the concept *PMDecorator*. A decorator object decorating $H$ is used to construct the plane map $H$ when reading from the input stream, or to explore the structure when printing to the output stream. We omit the implementation details.

We only present the I/O format that is similar to that used in LEDA for general graphs. There is a header and then three sections storing the objects vertices, halfedges, faces:

```
Plane_map_2
vertices n1
halfedges n2
faces n3
0    { isolated incident_object, mark, point }
...
n1-1 { isolated incident_object, mark, point }
0    { opposite, prev, next, vertex, face, mark }
...
n2-1 { opposite, prev, next, vertex, face, mark }
0    { halfedge, fclist, ivlist, mark }
...
n3-1 { halfedge, fclist, ivlist, mark }
```

there are $n_1$ lines for vertices, $n_2$ lines for halfedges, and $n_3$ lines for faces. All objects are indexed by non-negative integers. Vertex lines contain a boolean marker *isolated* followed by the index of an incident object (a face if *isolated* is true, otherwise the first halfedge of the adjacency list), the attribute and the embedding. Halfedge lines store the link structure (again by indices representing the objects): the *opposite* (also called twin or reversal) halfedge, the *prev*ious and *next* halfedge of its face cycle, the incident *vertex* and the incident *face*, and the attributed *mark*. The face lines have no fixed length as the number of face cycles and isolated vertices is not bounded. Both lists *fclist* (for face cycles) and *ivlist* (for isolated vertices) are white space separated lists of numbers. Their elements are the indices of one halfedge from the corresponding face cycle or the indices of the isolated vertices in the interior of the faces respectively. The *halfedge* is the index of a halfedge of the outer face cycle of the face and the *mark* is again the attribute of the face. Note that as our index range starts at 0, we code undefined references by $-1$.

I/O and cloning processes bear a strong similarity. In both processes one creates isomorphic representations of pointer structures. In case of output the representation of typed pointers (handles) is a unique numbering of all objects that can be translated back during an input process.

### PM_visualizor - drawing plane maps in a window

We offer a decorator drawing a plane map into a CGAL window stream, which is basically a LEDA window offering stream operations for all affine kernel objects of the CGAL geometry kernels. The class template *PM_visualizor*<*PMCDEC*, *GEOM*, *COLORDA*> requires models of the three template parameters for instantiation. The first two can be instantiated by *PM_const_decorator*<> and any geometry kernel being a model of the concept *AffineGeometryTraits_2*. The third parameter assigns colors and sizes to the objects of the plane map depending on their attributes by the following class concept:

```
struct COLORDA {
  CGAL::Color color(Vertex_const_handle, const Mark& m) const;
  CGAL::Color color(Halfedge_const_handle, const Mark& m) const;
  CGAL::Color color(Face_const_handle, const Mark& m) const;
  int width(Vertex_const_handle, const Mark& m) const;
  int width(Halfedge_const_handle, const Mark& m) const;
};
```

On construction the visualizor obtains a window stream $W$, a decorator $D$, a geometry kernel $K$, and a color data accessor $C$. The plane map referenced by $D$ is drawn in the window $W$ with the properties as specified by $C$.

```
PM_visualizor(CGAL::Window_stream& W, const PMCDEC& D,
              const GEOM& K, const COLORDA& C);
```

The class offers drawing by object or drawing of the full structure by the methods:

```
void draw(Vertex_const_handle v) const
void draw(Halfedge_const_handle e) const
void draw(Face_const_handle f) const
void draw_map() const
```

We do not show their implementation here. For the drawing of the faces we use the techniques that are used by LEDA windows to draw polygons.

## 4.7 Subdivision, Selection, and Simplification

In this section we present a software module for the overlay of segments and plane maps. We first give a formal introduction to the notions and difficulties concerning overlay and support. We then present the overlay calculation of a set of segments. We show how we use a generic sweep module to produce the 1-skeleton of the output plane map. In a different section we show how to add face objects to the 1-skeleton to complete the output structure. The second operation concerns the overlay of two plane maps. We use the same generic sweep module with slightly more elaborate adaptation to obtain again the 1-skeleton of the overlay. The face production phase will be the same as before. In case of the second overlay operation our sweep adds a transfer of information assigned to the objects of the two input plane maps to corresponding objects in the output structure. This allows us to use the module for binary set operations on plane map structures. Such set operations use a selection phase on the transferred information items. The selection phase is descibed below in an additional section. The last section in this document concerns structural simplification of the output plane map. We will see that there can be substructures in the output plane map that can be simplified without losing any information when the plane map is interpreted as a point set.

### 4.7.1 Notions and definitions

If we consider our overlay process as a transformation of input objects to output objects then we can define the support relation as follows.

**Definition 10 (support):** Consider an algorithm $T$ that transforms a set of input objects $A$ to a set of output objects $B$ where each $a \in A$ and $b \in B$ represents a subset of $R^2$. We say that *$a$ supports $b$* if $b$ is a subset of $a$ with respect to the represented point sets.

We will anchor this notion in the following.

**Overlay of a set of segments** For a segment $s = (p,q)$, $p = source(s)$, $q = target(s)$ and $p$, $q$ are called the endpoints of $s$. Let us consider $s$ as a disjoint union of its endpoints and its relative interior $\mathrm{relint}\, s$. A set of segments $S$ partitions the plane into cells of different dimensions. For each point $r \in \mathcal{R}^2$ it can happen that

(i) $r$ is equal to some endpoint $p$ of some segment $s$, or

(ii) $r$ is part of the relative interior of some segment $s$, or

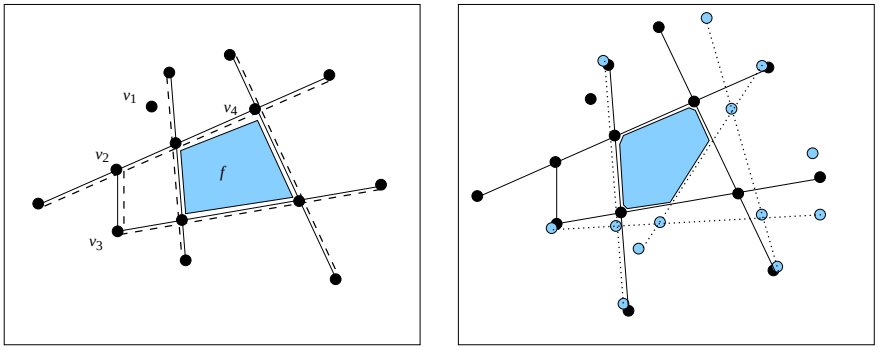



Figure 4.8: The overlay of a set of segments and of two plane maps. The left figure shows a set of dashed segments. $v_1$ is an isolated vertex, $v_2$ is an endpoint in the interior of another segment, $v_3$ is a vertex supported by two endpoints, $v_4$ is the intersection of the relative interior of two segments. The edges are drawn with solid line segments. One bounded face is greyed. The right figure shows the 1-skeleta of two plane maps. Degenerate situations are identical vertices, vertices in the interior of edges, and overlapping edges.

(iii) $r$ is not part of any segment at all.

Note that (i) and (ii) do not exclude each other. Now consider the geometric structure built by all segments. The *overlay* of all segments is the subdivision of all points in $\mathcal{R}^2$ with respect to the three criteria (i) to (iii) above including their topological neighborhood and the knowledge how parts of the segments in $S$ support the cells of the subdivision.

We store the overlay of $S$ in a plane map $P = (V,E,F)$ in the standard way. For each point $r$ in (i) there is a vertex $v$ in $V$ where the endpoint of the segment supports the vertex. If $r$ is additionally in the relative interior of some other segment according to (ii) then this segment also supports $v$. For each point in (ii) that is the unique intersection point of the relative interior of two segments (that do not overlap) there is a vertex in $V$ and the relative interior of each of the two segments supports that vertex. Between any two vertices in $V$ there is a uedge $e$ in $E$ if there is a segment $s$ that supports the straight line embedding of $e$ according to (ii) and there is no further vertex in the relative interior of $e$. The latter can happen for several segments that overlap. Any point of (iii) belongs to one of the maximal connected sets[10] of $\mathcal{R}^2 - S$ that form the faces of $P$ and is thus not supported by any segment at all.

**Overlay of two plane maps** Let $P_i = (V_i, E_i, F_i), i = 0, 1$ be two plane map structures. The overlay of two plane maps $P_0$, $P_1$ is the plane map $P$ representing the subdivision of the plane obtained by interpreting the skeleton objects of $P_i$ according to their embedding as trivial and non-trivial segments, constructing the overlay of these segments and adding the faces. To make this structure really helpful we explore the support relation between object of $P_i$ and $P$.

In general, each point $p$ in the plane is supported by that object of a plane map whose corresponding point set contains $p$. The support relation between $P_i$ and $P$ comes in two steps. Each 1-skeleton object of $P_i$ relates to the endpoint or relative interior of a segment that supports a skeleton object in $P$. Reversely, each object of $P$ (vertex, edge, or face) is supported by a unique supporting object in each of the two structures $P_i$ $(i = 0, 1)$. We show that this relation is well-defined.

**Lemma 4.7.1:** Any object of $P$ has exactly one supporting object in each of the $P_i$.

[10] path connected in the strong topological sense.

*Proof.* Obviously, each point of the plane is supported by an object of $P_i$. Therefore, we only have to argue why no two objects of $P_i$ can support one object of $P$. For vertices this is trivial. For a uedge $e$ in $P$ there can be only one uedge or one face of $P_i$ that supports $e$: assume that the embedding of $e$ covers points from more than one object of $P_i$. Then, $e$ either contains a vertex or crosses an edge in its interior. But then, the corresponding subdivision would have prevented the creation of $e$ in $P$ in the first place.

For a face $f$ of $P$ there can be only one face $f'$ of $P_i$ that supports $f$: assume otherwise, that $f$ contains points from different objects of $P_i$. As $f$ is an open connected point set it has to cover points of at least one boundary object from the 1-skeleton of $P_i$. But this object is part of the 1-skeleton of $P$ and can therefore never be part of $f$. □

In our implementation we determine the support relation in two phases. Any vertex $v$ in $V$ can be supported by a vertex $v_i$, a uedge $e_i$, or a face $f_i$ of $P_i$. If $v$ is supported by $v_i$ or $e_i$ we obtain this information in a plane sweep process. Assume that $v$ is supported by a face $f_i$ (then $v$ is supported by a vertex $v_{1-i}$). During the sweep process the determination of a support of $f_i$ is hard, as the face objects are not in reach. We determine $f_i$ in a postprocessing phase by a simple iteration over all vertices. Any edge $e$ in $E$ can be supported by a uedge $e_i$ or a face $f_i$ of $P_i$. A possible support by $e_i$ is handled during the sweep process. In case $e$ is part of a face $f_i$ (again $e$ is then supported by an edge $e_{1-i}$) we also determine $f_i$ in the postprocessing phase.

The support for a face $f$ in $F$ can be determined as follows. Assume that each directed edge $e$ in $E$ knows the faces $f_i$ supporting points in a small neighborhood on its left side ($i = 0, 1$). Then, $f$ can determine its two supporting faces $f_i$ via any edge in its boundary cycle. We will enrich the edges of $E$ by such support information and use it afterwards to transfer attributes from $f_i$ to $f$.

### 4.7.2 The class design

We start with the design of the class object. Our generic overlay class can be adapted via two interface concepts. We interface the underlying plane map via a plane map decorator *PMDEC*, we interface the underlying geometry via a geometry kernel *GEOM*. We inherit from *PMDEC* to obtain its interface methods.

⟨*PM overlayer*⟩≡
```
template <typename PM_decorator_, typename Geometry_>
class PM_overlayer : public PM_decorator_ {
  typedef PM_decorator_ Base;
  typedef PM_overlayer<PM_decorator_,Geometry_>  Self;
  const Geometry_& K; // geometry reference

public:
  typedef PM_decorator_                 Decorator;
  typedef typename Decorator::Plane_map Plane_map;
  typedef Geometry_                     Geometry;
  typedef typename Geometry::Point_2    Point;
  typedef typename Geometry::Segment_2  Segment;
  typedef typename Decorator::Mark      Mark;
  ⟨handles, iterators, and circulators from Decorator⟩
  ⟨info type to link edges and segments⟩
PM_overlayer(Plane_map& P, const Geometry& g = Geometry()) :
  Base(P), K(g) {}
```

⟨*subdivision*⟩
⟨*selection*⟩
⟨*simplification*⟩
⟨*helping operations*⟩
`}; // PM_overlayer<PM_decorator_,Geometry_>`

### 4.7.3 Overlay calculation of a list of segments

We want to calculate the plane map $P$ representing the overlay of a set $S$ of segments, some of which may be trivial. This task is basically split in **two phases**:

**overlay of segments** — the calculation of the 1-skeleton $P' = (V,E)$ of a plane map via the overlay of the segments in $S$ plus the calculation of a map *halfedge_below* : $V \to E$

**face creation** — the completion of the 1-skeleton $P'$ to a full plane map $P = (V,E,F)$ by creating all faces while using the information of the map *halfedge_below*.

For the overlay process we use the generic segment sweep module as presented in Section 4.8. There we presented a generic class *Segment_overlay_traits* realizing a generic sweep framework. To instantiate it we have to provide three components (input, output, geometry). In this instance the input is an iterator pair, the geometry is forwarded from the current class scope. Only for the output type we have to work a little more. We define a class *PMO_from_segs* that fits the output concept of *Segment_overlay_traits* and at the same time is a model for the *Below_info* concept required for the facet creation in Section 4.7.5. (See Figure 4.9.)

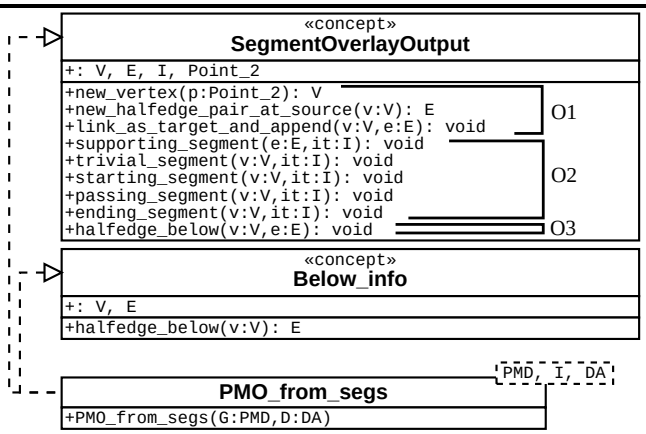



Figure 4.9: *PMO_from_segs* realizes the *Output* concept of the generic sweep module and the *Below_info* concept for the facet creation phase. In the figure *Vertex_handle*, *Halfedge_handle*, and *Iterator* have been replaced by the short symbols $V$, $E$, and $I$.

On creation an object of type *PMO_from_segs* references a plane map via a decorator $G$ and obtains a data accessor object $D$ of type *DA*. *PMO_from_segs* as a model of *SegmentOverlayOutput* triggers the correct update operations on the output plane map during the sweep. See the output concept in Figure 4.9. The method part O1 of *SegmentOverlayOutput* takes care of the plane map extension by new

vertices and edges. The part O2 allows to obtain information how the creation of the objects is linked to the input interators. In the implementation (which we do not show) of *PMO_from_segs* we forward this interface to methods of the data accessor of type DA. Finally, part O3 can be used to collect the additional information required for the facet creation. An edge *e* that is immediately below a vertex *v* is stored in the vertex object in a temporarily assigned data slot and can be retrieved after the sweep. *PMO_from_segs* as a *Below_info* model can thus afterwards deliver the halfedge *halfedge_below*$(v)$ for any vertex *v* of the plane map.

At this point our readers should take the module

```
generic_sweep< Segment_overlay_traits<PMO_from_segs<... >... >>
```

as a black box producing the 1-skeleton of $P$ with the properties required in Section 4.7.5. The specification of *Segment_overlay_traits* guarantees these properties of $P$ because *PMO_from_segs* fits the requirements of the output concept of *Segment_overlay_traits*.

Now, the overlay creation is trivial. Just create an output decorator object *Out* working on the plane map maintained by *PM_overlayer* and plug it into the segment sweep overlay framework *Segment_overlay_traits*. The used geometry is just forwarded from *PM_overlayer*. The *create* method of *PM_overlayer* is parameterized by the iterator type *Forward_iterator* and the data accessor class *Object_data_accessor*.

Note that the *halfedge_below* information collected during the sweep is associated with the vertices of the output map. The corresponding object *Out* triggers the output creation during the sweep and provides the halfedge-below information for the face creation in *create_face_objects*( ). *Out.clear_temporary_vertex_info*( ) just discards the temporarily allocated information slots (internally assigned to the vertices) on the heap.

⟨*subdivision*⟩≡

```
template <typename Forward_iterator, typename Object_data_accessor>
void create(Forward_iterator start, Forward_iterator end,
            Object_data_accessor& A) const
{
  typedef PMO_from_segs<Self,Forward_iterator,Object_data_accessor>
    Output_from_segments;
  typedef Segment_overlay_traits<
    Forward_iterator, Output_from_segments, Geometry> seg_overlay;
  typedef generic_sweep< seg_overlay > seg_overlay_sweep;
  typedef typename seg_overlay::INPUT input_range;
  Output_from_segments Out(*this, A);
  seg_overlay_sweep SOS( input_range(start, end), Out, K);
  SOS.sweep();
  create_face_objects(Out);
  Out.clear_temporary_vertex_info();
}
```

We summarize the calculated overlay properties and anticipate the costs of face creation in Section 4.7.5 and of the plane sweep description. (see Lemma 4.8.3).

**Lemma 4.7.2:** Assume that $S = set[start, end)$ is a set of segments and $A$ is a data accessor with the required methods (of constant cost). Then, $create(start, end, A)$ constructs in $P = (V, E, F)$ the overlay plane map of $S$. Let $n$ be the number of segments in $S$, $n_v = |V|$, $n_e = |E|$, and $\bar{n}_e$ the sum

of the support multiplicity of each edge over all edges. Then the runtime of the overlay process is dominated by the plane sweep and is therefore $O(n_v + \bar{n}_e + (n+n_v)\log(n+n_v))$.

### 4.7.4 Overlay calculation of two plane maps

We calculate the overlay $P$ of two plane maps $P_0$ and $P_1$. Both input structures are two correctly defined plane maps including incidence, geometric embedding, and markers. In the following we use the index $i = 0,1$ showing a reference to $P_i = (V_i, E_i, F_i)$; non-indexed variables refer to $P$.

The 1-skeleta of the two maps $P_0$ and $P_1$ subdivide the edges and faces of the complementary structures into smaller units. This means vertices and edges of $P_i$ can split edges of $P_{1-i}$ and face cycles of $P_i$ subdivide faces of $P_{1-i}$. The 1-skeleton $P' = (V,E)$ of $P$ is defined by the overlay of the embedding of the 1-skeleta of $P_0$ and $P_1$ (Take a trivial segment for each vertex and a segment for each edge and use the overlay definition of a set of segments above). Additionally, we require that $P'$ has the correct order in each adjacency list such that it is order-preserving regarding the embedding of the vertices.

Finally, the faces of $P$ refer to the maximal connected open point sets of the planar subdivision implied by the embedding of $P'$. The construction of the faces $F$ from $P'$ is described in Section 4.7.5. Each object $u$ from the output tuple $(V,E,F)$ has a *supporting* object $u_i, i = 0,1$ in each of the two input structures. Imagine the two maps to be transparencies, stacked one on top of the other. Then each point of the plane is covered by an object from each of the input structures. We analyse the support relation from input to output in order to transfer the attributes from $u_i$ to $u$.

According to our specification each object $u_i$ of $P_i$ carries an attribute[11] $mark(u_i)$ ($mark : (V_i \cup E_i \cup F_i) \to Mark$). We associate this information with the output object $u$ by $mark(u,i)$ (an overloaded function $mark : (V \cup E \cup F) \times \{0,1\} \to Mark$). This two-tuple of information per object can then be processed by some combining operation to a single value $mark(u)$ lateron.

We fix the following **input properties** for our structures $P_i$. Both plane maps $(V_i, E_i, F_i)$ consist of vertices, edges, and faces whose topology is accessible by our plane map interface and additionally each object $u_i$ carries an attribute $mark(u_i)$. The plane maps have an *order-preserving* embedding and their adjacency lists have a *forward prefix*. Actually we do not use this property of the input plane maps at this point but it is a general invariant of our plane map structures that makes some intermediate actions more efficient. The overlay process consists of **three phases**: The 1-skeleton $P'$ is produced by segment overlay. Afterwards we create the face objects. Finally, we analyse the support relation and transfer the marks of the input objects to the output objects.

**overlay of segments** — We use our generic segment overlay framework to calculate the overlay of a set of segments $S$. The set $S$ consists of all segments that are the embedding of edges in $E_i$ and additionally trivial segments representing all isolated vertices in $V_i$. The output structure $P' = (V,E)$ of the sweep phase is just the 1-skeleton of the output plane map $P$, but of course including an order-preserving embedding and a forward-prefix in the adjacency lists. The objects of the 1-skeleton carry additional structural information:

I1. Each vertex $v$ in $V$ knows a halfedge $e \in E : e = halfedge\_below(v)$ which is determined by the property that a vertical ray shot from $v$ along the negative $y$-axis hits $e$ first. Degeneracies are broken with a perturbation scheme: during the ray shooting all edges include their source vertex.

[11] we use a general attribute set, though with respect to Nef polyhedra $Mark := \{true, false\}$.

I2. For each object $u \in V \cup E$ there is a mapping to the supporting 1-skeleton objects of the input structures. The support information is incomplete with respect to face support.

**face creation** — The next phase after the sweep has to complete the plane map $P$. We basically have to create the face objects and construct their incidence structure. The face creation is done as presented in Section 4.7.5 and uses only I1.

**attribute transfer** — The final transfer of marks uses the embedding of the vertex list of $P$ and the additional information I1 and I2 to define $mark(u,i)$ for all objects $u$ in $P$.

⟨*subdivision*⟩+≡
```
void subdivide(const Plane_map& P0, const Plane_map& P1) const
{
  Const_decorator PI[2];
  PI[0] = Const_decorator(P0); PI[1] = Const_decorator(P1);
  ⟨fi lling the input segment list⟩
  ⟨sweeping the segments and creating the faces⟩
  ⟨transfering the marks of supporting objects⟩
}
```

### Temporary information associated with objects

We have to associate temporary information with the objects of the output plane map. In this section we abstractly use sets in a pseudo code notation to underline the origin of plane map objects. The objects from these sets are realized by the corresponding handle types (and therefore their type does not allow to mark their origin). Undefined objects are detectible via default handles.

At first we interpret the input 1-skeleta geometrically. We collect a set of trivial and non-trivial segments $S$. For each edge in $E_i$ we add a non-trivial segment to $S$ and for each isolated vertex of $V_i$ we add a trivial segment to $S$. We store the origin of the objects in $S$ via a function

$$\begin{aligned} \mathit{From} &: S \to (V_{0,1} \cup E_{0,1}) \times \{0,1\} \\ \mathit{From}(s) &= \begin{cases} (v_i,i) & \text{if } s \text{ is a trivial segment refering to an isolated vertex } v_i \text{ from } P_i, \\ (e_i,i) & \text{if } s \text{ is a non-trivial segment refering to an edge } e_i \text{ from } P_i. \end{cases} \end{aligned}$$

*From* is implemented as a hash map *From* whose domain are iterators (with value type segment) and whose value is a structure *Seg_info* with members *v, e, i* storing the above pairs.

⟨*info type to link edges and segments*⟩+≡
```
typedef std::list<Segment>                    Seg_list;
typedef typename Seg_list::const_iterator     Seg_iterator;
typedef std::pair<Seg_iterator,Seg_iterator> Seg_it_pair;
```

In the first phase we fill the segment input list with a non-trivial segment underlying each edge and with a trivial segment for each isolated vertex of the two input structures. Additionally, we store

hashed links from the iterators to the edges/vertices to store their origin.

⟨*filling the input segment list*⟩≡
```
Seg_list Segments; int i;
CGAL::Unique_hash_map<Seg_iterator,Seg_info> From;
for (i=0; i<2; ++i) {
  Vertex_const_iterator v;
  for(v = PI[i].vertices_begin(); v != PI[i].vertices_end(); ++v)
    if ( PI[i].is_isolated(v) ) {
      Segments.push_back(segment(PI[i],v));
      From[--Segments.end()] = Seg_info(v,i);
    }
  Halfedge_const_iterator e;
  for(e = PI[i].halfedges_begin(); e != PI[i].halfedges_end(); ++e)
    if ( is_forward_edge(PI[i],e) ) {
      Segments.push_back(segment(PI[i],e));
      From[--Segments.end()] = Seg_info(e,i);
    }
}
```

During the sweep phase we collect additional information in temporary information containers associated with the objects $u \in V \cup E \cup F$ of $P$.

*assoc_info*$(u)$ creates the temporary object on the heap
*discard_info*$(u)$ discards the object and frees the memory

Within these objects we store the following pair of mark attributes (indexed by $i$):

*Mark mark*$(u,i)$ for $i = 0,1$ and $u \in V \cup E \cup F$

For each vertex $v$ we collect *skeleton support* information.

$V_i$ *supp_vertex*$(Vv, int i)$ the vertex from $V_i$ supporting $v$ if it exists, else undefined.
$E_i$ *supp_halfedge*$(Vv, int i)$ the edge from $E_i$ supporting $e$ if it exists, else undefined.

And for each edge $e$ we want to know

$E_i$ *supp_halfedge*$(E\ e, int\ i)$ the edge from $E_i$ supporting $e$ if it exists, else undefined.
*Mark incident_mark*$(E\ e, int\ i)$ the mark of the face from $P_i$ supporting a small neighborhood left of $e$.

The information is collected during the sweep phase by a corresponding model of the output concept used in our generic sweep framework. We omit the realization of the above attribution. Details can be found in the accompanying research report.

### The sweep instantiation

We have to provide the three components (input, output, geometry) necessary to instantiate the traits model *Segment_overlay_traits* for our generic plane sweep framework. The input is an iterator pair, the geometry is forwarded from the current class scope. Only for the output type we have to work a little more. We define a class *PMO_from_pm* below which allows us to track the support relationship from input objects (segments handled via iterators) to the output objects (vertices and halfedges) via

the call-back methods triggered during the sweep. Please refer to the description of *Segment_overlay_traits*.

The methods of *PMO_from_pm* fit the output concept requirements of *Segment_overlay_traits*. The functionality is such that the skeleton is created and the support information is associated with the newly created objects. *PMO_from_pm* is a class template on the global implementation scope as the usage of local class types (within the scope of *PM_overlayer*) is not allowed by some current C++ compilers.

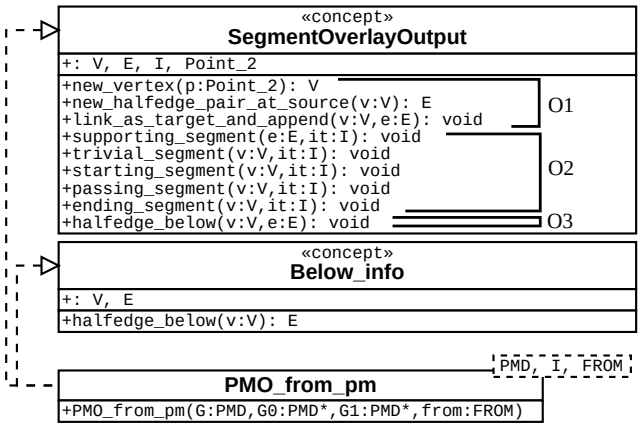



Figure 4.10: *PMO_from_pm* realizes the *Output* concept of the generic sweep module and the *Below_info* concept for the facet creation phase. In the figure *Vertex_handle*, *Halfedge_handle*, and *Iterator* have been replaced by the short symbols $V$, $E$, and $I$.

We shortly describe how the temporary information associated with the skeleton objects is retrieved from the use of *PMO_from_pm* (cf. Figure 4.10) as an output model in the generic sweep. The operations in section O1 create and link new objects in $P$ and additionally use *assoc_info*( ) to create the temporary storage containers. The operations in section O2 accumulate the support information. As an example, we show how the information for *supp_halfedge*$(V, int)$ is collected:

```
void PMO_from_pm<PMD,I,FROM>::passing_segment(V v, I it)
{ int i = From[it].i;
  supp_halfedge(v,i) = From[it].e;
}
```

At each event of the sweep, a vertex $v$ is created in $P$ and if $v$ lies in the interior of the segment $*it$ the above operation is called. Thereby, after the sweep all edges from $E_i$ that support a vertex $v$ from $V$ are associated with $v$. The remaining support is determined similarily. The operation in section O3 is realized to collect information that allows *PMO_from_pm* to serve as a model for the *Below_info* concept.

Now, creating the overlay is a trivial plugging of types into the generic plane sweep framework, a creation of the sweep object with input, output and geometry references, and a final execution of the sweep. Afterwards, the faces are created.

⟨*sweeping the segments and creating the faces*⟩≡

```
typedef PMO_from_pm<Self,Seg_iterator,Seg_info> Output_from_plane_maps;
typedef Segment_overlay_traits<
  Seg_iterator, Output_from_plane_maps, Geometry> pm_overlay;
```

```
typedef generic_sweep< pm_overlay > pm_overlay_sweep;
Output_from_plane_maps Out(*this,&PI[0],&PI[1],From);
pm_overlay_sweep SOS(Seg_it_pair(Segments.begin(),Segments.end()),Out,K);
SOS.sweep();
create_face_objects(Out);
```

### Transfering the marks

After the sweep and the face creation the input for this phase is a plane map $P = (V,E,F)$ enriched by additional information attributed to the 1-skeleton objects of $P$. The output vertices in $V$ are linked to their supporting skeleton input objects (vertices and edges). The output edges in $E$ are linked to their supporting input edges. The support knowledge with respect to input faces is still missing. In the following we analyse this support but do not store it explicitly. Instead we only transfer the marks. There are several properties of the constructed subdivision $P$ which help us to do this.

- the vertices are constructed in the order of the sweep. By iterating them in their construction order we can rely on the fact that we iterate according to the lexicographic order of their embedding.
- the halfedges out of a vertex $v$ are ordered around $v$ counterclockwise (with respect to the embedding of their target). We can therefore use a forward iteration to propagate face information from bottom to top (on forward oriented edges).
- the first face *faces_begin*( ) in the list of all faces is the unbounded face. This holds for $P$, $P_0$, and $P_1$.

⟨*transfering the marks of supporting objects*⟩≡

```
⟨initialize the outer face object⟩
Vertex_iterator v, vend = vertices_end();
for (v = vertices_begin(); v != vend; ++v) {
  ⟨determine mark of face below v⟩
  ⟨complete marks of vertex v⟩
  ⟨handle all forward oriented edges starting in v⟩
}
⟨transfer the marks to face objects⟩
```

The transfer of face support marks is based on the following fact.

**Fact 2:** Let $p$ be a point of the plane not part of the 1-skeleton of $P_i$, $q$ be a point within the unbounded face of $P_i$, and ρ be any curve from $p$ to $q$ not containing any vertex of $P_i$. Assume ρ intersects an edge $e$ of the 1-skeleton of $P_i$ and let $e$ be the first such edge when following ρ from $p$ to $q$. Then, $p$ is part of the face incident to $e$. If ρ does not intersect the 1-skeleton, then $p$ is part of the unbounded face of $P_i$.

The above fact is a consequence of the connectedness property of the faces of $P_i$. We now consider point $p$ as part of $P$. For $p$ we consider a special path ρ as depicted in Figure 4.11. We walk down along a vertical ray (in direction of the negative $y$-axis). If we cross a bundle of edges incident to a vertex $v$ the path turns just below the lowest edge and follows the lowest edge in parallel until it is just below $v$. We iterate this construction until it ends in a point $q$ in the unbounded face of $P$. Each

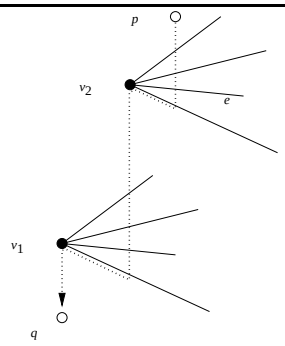



Figure 4.11: The face support iteration unrolled. We examine a position $p$ within the plane map $P$ and try to find the support by a face $f_i$ in the plane map $P_i$. We have two vertices $v_1$ and $v_2$ from $V$ with their forward-oriented edge bundle. The face that supports $p$ with respect to $P_i$ can be determined by following the dotted path until the outer unbounded face of $P_i$ is reached.

edge $e$ that is crossed by $\rho$ is supported by an edge either from $P_i$ or from $P_{1-i}$. In the former case the first such edge determines the face $f_i$ supporting $p$. If there is no such edge then $p$ is supported by the unbounded face of $P_i$. We want to determine face support for many vertices and edges, thus we do not want to pay such a walk for each query point $p$. Instead we associate with each edge $e$ face support knowledge in the two slots $mark(e,i)$ and $incident\_mark(e,i)$. The idea is that these slots store the knowledge obtained from a reversal walk from $q$ to $p$. Whenever our path $\rho$ crosses an edge $e$ in $P$ that is supported by an edge $e_i$ in $P_i$, then we associate the mark knowledge plus the mark of the supporting faces from $P_i$ (of a small neighborhood left and right of $e$) with $e$. If the information is already constructed for all edges below a query point $p$, we can obtain the support information in constant time.

We now come to the coding. We want to complete the support marks for a vertex $v$ and the edges $e$ of the adjacency list of $v$ that are forward oriented[12]. Consider to follow $\rho$ reversely with a pen starting in $q$. Then $m\_below[2]$ always stores the marks of the faces of $P_i$ that support the position of the pen. In the beginning $m\_below[i]$ stores the mark of the face $f_i$ below $v$. Note that we obtain both marks for $i = 0,1$ either from the outer input faces surrounding the plane maps $P_i$ or from the halfedge below $v$. If *e_below* exists then it was already treated as a forward oriented edge of a vertex already handled in the vertex iteration.

⟨*initialize the outer face object*⟩≡
```
Face_iterator f = faces_begin(); assoc_info(f);
for (i=0; i<2; ++i) mark(f,i) = PI[i].mark(PI[i].faces_begin());
```

Note that the iteration over all vertices $v$ has the invariant that either $v$ has no halfedge below, or if it has a halfedge *e_below* then *e_below* has all marks correctly assigned ($mark(e\_below,i)$ and $incident\_mark(e\_below,i)$ are set for both $i = 0,1$). Note that each vertex $v$ of $P$ knows the halfedge below it, thus the face support marks can be initialized in constant time.

[12]Backward oriented edges have forward oriented twins.

⟨*determine mark of face below v*⟩≡
```
Halfedge_handle e_below = halfedge_below(v);
Mark m_below[2];
if ( e_below != Halfedge_handle() ) {
  for (int i=0; i<2; ++i) {
    m_below[i] = incident_mark(e_below,i);
  }
} else { // e_below does not exist
  for (int i=0; i<2; ++i)
    m_below[i] = PI[i].mark(PI[i].faces_begin());
}
```

If the vertex $v$ is not supported by a skeleton object of $P_i$ then it is supported by a face. We obtain the mark of the face from *m_below* in this case.

⟨*complete marks of vertex v*⟩≡
```
for (i=0; i<2; ++i)
  if ( supp_halfedge(v,i) != Halfedge_const_handle() ) {
    mark(v,i) = PI[i].mark(supp_halfedge(v,i));
  } else if ( supp_vertex(v,i) != Vertex_const_handle() ) {
    mark(v,i) = PI[i].mark(supp_vertex(v,i));
  } else {
    mark(v,i) = m_below[i];
  }
```

We have to complete the mark information for all edges of $P$. We do the job for all forward oriented edges in the adjacency list of each vertex $v$. How does a halfedge $e$ of $P$ obtain mark information with respect to the two input structures $P_i$? We just have to determine the supporting objects (edge or face) from each of both. It is either supported by two overlapping edges $e_0, e_1$ or only supported by one edge $e_i$ and one face $f_{1-i}$. Note that a supporting edge $e_i$ allows access to its mark and to the two faces incident to it and its twin. The supporting edge $e_i$ of $e$ can be obtained via *supp_halfedge*$(e,i)$. If $e$ is not supported by an edge in $P_i$ then the mark of the input face can be obtained from *m_below*$[i]$. Each supporting input edge $e_i$ of $e$ changes *m_below*$[i]$ for the next output edge in the bundle iteration. If $e$ is not supported by an edge in $P_i$ then the supporting face determines the mark of $e$ and the two *incident_mark* entries. The invariant for all edges $e$ in the iteration below is: if $e$ is not supported by an edge $e_i$ of $P_i$ then *m_below*$[i]$ contains the mark of the face supporting $e$ in $P_i$.

⟨*handle all forward oriented edges starting in v*⟩≡
```
if ( is_isolated(v) ) continue;
Halfedge_around_vertex_circulator
  e(first_out_edge(v)), hend(e);
CGAL_For_all(e,hend) {
  if ( is_forward(e) ) {
    Halfedge_const_handle ei;
    bool supported;
    for (int i=0; i<2; ++i) {
      supported = ( supp_halfedge(e,i) != Halfedge_const_handle() );
      if ( supported ) {
        ei = supp_halfedge(e,i);
        incident_mark(twin(e),i) =
```

```
          PI[i].mark(PI[i].face(PI[i].twin(ei)));
        mark(e,i) = PI[i].mark(ei);
        incident_mark(e,i) = m_below[i] =
          PI[i].mark(PI[i].face(ei));
      } else { // no support from input PI[i]
        incident_mark(twin(e),i) = mark(e,i) = incident_mark(e,i) =
          m_below[i];
      }
    }
  } else break;
}
```

The last chunk of this section transfers the support marks to the face object. For all bounded faces $f$ we just transfer the marks from the bounding face cycle to the face. As all edges $e$ carry the $incident\_mark(e,i)$ attribute this completes the structure.

⟨*transfer the marks to face objects*⟩≡

```
for (f = ++faces_begin(); f != faces_end(); ++f) { // skip first face
  assoc_info(f);
  for (i=0; i<2; ++i) mark(f,i) = incident_mark(halfedge(f),i);
}
```

We can now summarize the calculated overlay properties, we anticipate the costs of face creation as described in the next section and the analysis of the sweep description in Lemma 4.8.3.

**Lemma 4.7.3:** Assume that $P_0$ and $P_1$ are plane maps whose embedding is order-preserving and the adjacency lists have a forward prefix, then $subdivide(P_0,P_1)$ constructs in $P=(V,E,F)$ the overlay plane map of $P_0$ and $P_1$ and each object $u \in V \cup E \cup F$ carries the mark information $mark(u,i)$ from the corresponding supporting object of the input plane map.

Let $n_i$ be the size of $P_i$ and $n$ be the size of $P$. Then the runtime of the overlay process is dominated by the plane sweep of the skeleton objects of $P_0$ and $P_1$ and is therefore $O((n_0+n_1+n)\log(n_0+n_1+n))$.

### 4.7.5 Creating face objects

Input to this section is the 1-skeleton of a plane map $P'=(V,E)$ whose embedding is *order-preserving* and whose adjacency lists have a *forward-prefix*. The objective of this section is to *create the face objects* that complete $P'$. The correct output structure $P=(V,E,F)$ of this section is a plane map with the property that there are face objects $f$ in $F$ corresponding to maximal connected point sets which are a result of the partitioning of the plane by the 1-skeleton $P'$. All faces are defined via their bounding face cycles. Each face object has one halfedge link into the one unique outer face cycle (if existing), a list of halfedges each of which represents interior hole face cycles and a list of isolated vertices that represent trivial face cycles. To assign face cycles to face objects we need to know two properties of the plane map skeleton:

- for each face cycle we need to know if it is an outer face cycle or a hole face cycle.
- for two face cycles *fc1* and *fc2* we need to know if we can connect them by a path in the plane which does not cross any other face cycle.

We adapt an idea from [dBvKOS97]. The path connectivity making disjoint face cycles bounding the same face, can be modeled by a vertical visibility graph of the minimal vertices[13] of each face cycle. We create faces and assign face cycles based on this property and transfer $P'$ to $P$ thereby.

Let $C$ be a set of face cycles of the plane map skeleton. For each face cycle $c$ let $\mathit{MinimalHalfedge}[c]$ be the halfedge $e$ whose target vertex has minimal coordinates (lexicographically). Let $\mathit{FaceCycle}[e]$ be the face cycle containing $e$. We examine the following implicitly defined graph $G$. Each face cycle of $P'$ is a node of $G$. Let us link two face cycles $c_1$ and $c_2$ by an undirected edge of $G$ if $\mathit{target}(\mathit{MinimalHalfedge}[c_1])$ has a vertical view down to an edge of $c_2$ (in $P$). Note that face cycles consist of halfedges and thus we have to refer to the correct one of the two paired halfedges respecting the embedding when looking at face cycles (our faces are left of the directed halfedges, thus consider the bidirected twins to be separated by an infinitesimal distance, then the visibility is uniquely defined). Note that the embedding of a face cycle $c$ at its minimal halfedge gives us the criterion to separate outer face cycles and hole face cycles. Whenever the underlying line segments of $e = \mathit{MinimalHalfedge}[c]$ and $\mathit{next}(e)$ form a left turn $c$ is an outer face cycle. When they form a right turn the vertex $\mathit{target}(e)$ has a free view down and thus $e$ belongs to a hole.

Note that we do not explicitly model the visibility graph. Instead the recursive behavior of the operation *determine_face*( ) used below imitates a DFS walk on the visibility graph. In the following method we have the vertical visibility coded via a data accessor $D$ providing for all vertices $v \in V$ the knowledge about the halfedge below $v$. $D.\mathit{halfedge\_below}(v)$ either provides the halfedge of $E$ that is hit first by a vertical ray downwards or an uninitialized halfedge if there is none.

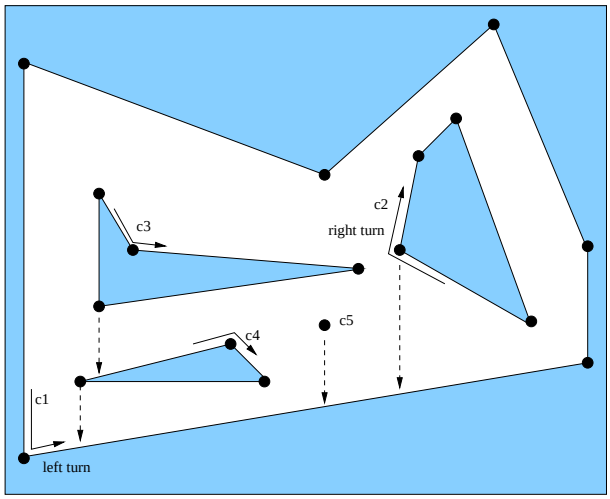



Figure 4.12: Face cycles bounding a face. c1 is the outer face cycle, c2, c3, and c4 are hole cycles, c5 is an isolated vertex. The minimal vertices of each face cycle are the origins of the dashed vertical arrows down.

The following template type parameter *Below_info* has to fit the concept *Below_info* of the Figures 4.9 and 4.10.

[13] minimal with respect to the lexicographic order of the point coordinates of their embedding

⟨*helping operations*⟩+≡
```
template <typename Below_info>
void create_face_objects(const Below_info& D) const
{
  CGAL::Unique_hash_map<Halfedge_handle,int> FaceCycle(-1);
  std::vector<Halfedge_handle>  MinimalHalfedge;
  ⟨link halfedges to face cycles and determine minimal halfedges⟩
  ⟨create face objects for outer face cycles and create links⟩
  ⟨link holes and isolated vertices to face objects⟩
}
```

We iterate all halfedges and assign a number for each face cycle. After the iteration for a halfedge $e$ the number of its face cycle is $FaceCycle[e]$ and for a face cycle $c$ we know $MinimalHalfedge[c]$.

⟨*link halfedges to face cycles and determine minimal halfedges*⟩≡
```
int i=0;
Halfedge_iterator e, eend = halfedges_end();
for (e=halfedges_begin(); e != eend; ++e) {
  if ( FaceCycle[e] >= 0 ) continue; // already assigned
  Halfedge_around_face_circulator hfc(e),hend(hfc);
  Halfedge_handle e_min = e;
  CGAL_For_all(hfc,hend) {
    FaceCycle[hfc]=i; // assign face cycle number
    if ( K.compare_xy(point(target(hfc)), point(target(e_min))) < 0 )
      e_min = hfc;
  }
  MinimalHalfedge.push_back(e_min); ++i;
}
```

We now know the number of face cycles $i$ and we have a minimal halfedge $e$ for each face cycle. We just check the geometric embedding of $e$ and $next(e)$ to characterize the face cycle (outer or hole). Note that the two edges cannot be collinear due to the minimality of $e$ (the lexicographic minimality of the embedding of its target vertex). Outer face cycles obtain face objects right away. Hole cycles whose *halfedge_below* information is undefined are associated with the unique outer face. After this chunk *f_outer* is the first face object *faces_begin*( ) in the list of all face objects, and all outer face cycles have face objects with temporary mark information slots expanded.

⟨*create face objects for outer face cycles and create links*⟩≡
```
Face_handle f_outer = new_face();
for (int j=0; j<i; ++j) {
  Halfedge_handle e = MinimalHalfedge[j];
  Point p1 = point(source(e)),
        p2 = point(target(e)),
        p3 = point(target(next(e)));
  if ( K.leftturn(p1,p2,p3) ) { // leftturn => outer face cycle
    Face_handle f = new_face();
    link_as_outer_face_cycle(f,e);
  }
}
```

Now, the only halfedges not linked are those on hole face cycles. We use a recursive scheme to find the bounding cycle providing the face object and finally iterate over all isolated vertices to link them accordingly to their containing face object. Note that in this final iteration all halfedges already have face links. This ensures termination. The recursive operation *determine_face*$(e,\dots)$ returns the face containing the hole cycle of $e$ (see the specification in the next section). As a postcondition of this chunk we have all edges and isolated vertices linked to face objects, and all face objects know their bounding face cycles.

⟨*link holes and isolated vertices to face objects*⟩≡
```
for (e = halfedges_begin(); e != eend; ++e) {
  if ( face(e) != Face_handle() ) continue;
  Face_handle f = determine_face(e,MinimalHalfedge,FaceCycle,D);
  link_as_hole(f,e);
}
Vertex_iterator v, v_end = vertices_end();
for (v = vertices_begin(); v != v_end; ++v) {
  if ( !is_isolated(v) ) continue;
  Halfedge_handle e_below = D.halfedge_below(v);
  if ( e_below == Halfedge_handle() )
    link_as_isolated_vertex(f_outer,v);
  else
    link_as_isolated_vertex(face(e_below),v);
}
```

When we call *determine_face*$(e,\dots)$ we know that the halfedge $e$ is not yet linked to a face object and thus, no halfedge in its face cycle is linked. Thus we jump to the minimal halfedge and look down. If we see nirvana then we have to link the unlimited face *f_outer*. If we see a halfedge we ask for its face. If it does not have one we recurse. Note that the target vertex of the minimal halfedge actually has a view downwards as we examine a hole face cycle. The method *link_as_hole* does the linkage between the face object and all edges of the face cycle. Its cost is linear in the size of the face cycle. Note also that we do the linking bottom up along the recursion stack for all visited hole cycles. Thus, we visit each hole face cycle only once as afterwards each edge of the face cycle is incident to a face.

Look at our example in Figure 4.12. When *determine_face* is called for an edge $e$ of face cycle c3, then the procedure first finds an edge of c4. If c4 was not visited yet by an earlier call, then the method recurses to c4 before it finds the correct face object via the outer face cycle c1.

⟨*helping operations*⟩+≡
```
template <typename Below_info>
Face_handle determine_face(Halfedge_handle e,
  const std::vector<Halfedge_handle>& MinimalHalfedge,
  const CGAL::Unique_hash_map<Halfedge_handle,int>& FaceCycle,
  const Below_info& D) const
{
  Halfedge_handle e_min = MinimalHalfedge[FaceCycle[e]];
  Halfedge_handle e_below = D.halfedge_below(target(e_min));
  if ( e_below == Halfedge_handle() ) // below is nirwana
    return faces_begin();
  Face_handle f = face(e_below);
  if (f != Face_handle()) return f; // has face already
  f = determine_face(e_below, MinimalHalfedge, FaceCycle,D);
```

```
    link_as_hole(f,e_below);
    return f;
  }
```

The explanations of the recursion condition of *determine_face* should convince you that:

**Lemma 4.7.4:** Assume that $P'$ is the 1-skeleton of a plane map whose embedding is order-preserving and the adjacency lists have a forward prefix. Let additionally all vertices know the halfedge visible along a vertical ray shot down, then *create_face_objects*( ) completes $P$ as a plane map with runtime *linear* in the size of the 1-skeleton $P$'.

### 4.7.6 Selecting marks

For the selection we just iterate over all objects, read the marks refering to the two input structures, apply our selection operation, and store the mark back into the object. At this place, we discard the additional information which was accumulated during the subdivision. The flexibility of the operation is achieved by a template type parameter *Selection*. An object *predicate* of type *Selection* must provide a binary function operator returning a new mark object. The runtime of the selection phase is obviously linear in the size of the plane map $P$. The method *discard_info* just discards the temporarily allocated information containers associated to the objects.

⟨*selection*⟩≡

```
  template <typename Selection>
  void select(Selection& predicate) const
  {
    Vertex_iterator vit = vertices_begin(),
                    vend = vertices_end();
    for( ; vit != vend; ++vit) {
      mark(vit) = predicate(mark(vit,0),mark(vit,1));
      discard_info(vit);
    }
    Halfedge_iterator hit = halfedges_begin(),
                      hend = halfedges_end();
    for(; hit != hend; ++(++hit)) {
      mark(hit) = predicate(mark(hit,0),mark(hit,1));
      discard_info(hit);
    }
    Face_iterator fit = faces_begin(),
                  fend = faces_end();
    for(; fit != fend; ++fit) {
      mark(fit) = predicate(mark(fit,0),mark(fit,1));
      discard_info(fit);
    }
  }
```

Note that after this phase the plane map output has again the input properties of the overlay calculation operation from Section 4.7.4.

**Lemma 4.7.5:** The selection phase has runtime linear in the size of the plane map.

### 4.7.7 Simplification of attributed plane maps

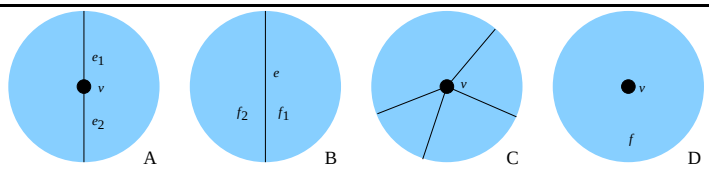



Figure 4.13: The possible configurations for simplification.

In this section we examine the task to simplify a given plane map to reach a minimimal representation (minimimal number of objects of the plane map structure, while the underlying attributed point set stays the same). There are three situations where one can imagine to simplify the structure (see Figure 4.13):

1. A vertex $v$ which is incident to two edges $e_1$, $e_2$ both supported by the same line where all three objects have the same mark can be unified into one edge without changing the stored point set. (Figure 4.13,A)

2. A uedge $e$ which has the same mark as the two faces $f_1$ and $f_2$ incident to it does not contribute any structural information and thus can be removed (Figure 4.13,B).

3. A vertex $v$ where all the edges of its adjacency list and also all incident faces have the same mark as the vertex also carries no structural information (Figure 4.13,C,D).

Note that the simplification configurations map to the properties of the local views of Nef faces in Lemma 4.3.1. If we first remove edges of the second case then the vertices of case three have no incident edges at all and thus can be easily identified as isolated vertices whose surounding face has the same mark. The first case does only play a role if one of the faces incident to the edge carries a different mark than the edge.

We can thus easily formulate the simplification routine. However, there are some problems with the update operations of the plane map structure. How can we maintain the face objects and incidence links to halfedges and vertices if we are unifying faces by deleting edges? The trivial way does not work within our time bound. We cannot afford to maintain the face objects in a correct status in each step of the simplification, as this would mean to repeatedly iterate total face cycles.

Note that we cannot just discard all faces and recreate them using a similar scheme as the one based on the *halfedge_below* information due to the fact that referenced edges might be deleted in the simplification process. Thereby, face creation as described in Section 4.7.5 is not possible without a new sweep. We take a different approach. We use a unification history stored in a partition data structure instead of the geometrically defined *halfedge_below* information as a criterium for linking face cycles to face objects.

All face cycles (edges and isolated vertices) reference face objects. When we have to unify two different faces due to the deletion of an edge separating them, we store this fact by a union operation in a partition data structure. The face that is finally assigned to all the face cycles of the faces in one block is the one associated with the canonical item of the block (obtained by the find operation).

⟨*simplifi cation*⟩≡
```
template <typename Keep_edge>
void simplify(const Keep_edge& keep) const
{
  typedef typename CGAL::Partition<Face_handle>::item partition_item;
  CGAL::Unique_hash_map<Face_iterator,partition_item> Pitem;
  CGAL::Partition<Face_handle> FP;
  ⟨initialize blocks corresponding to faces⟩
  ⟨simplify via non-separating halfedges⟩
  ⟨recollect face cycles per blocks⟩
  ⟨simplify via vertices⟩
  ⟨remove superfluous face objects⟩
}
```

We assign one partition item to each face object and make the item accessible to the face via a hash map. During the assignment of face cycles to face objects we will only use links from skeleton objects like vertices and edges to faces. We therefore can discard all face cycle entries in the faces (the links from face objects to skeleton objects).

⟨*initialize blocks corresponding to faces*⟩≡
```
Face_iterator f, fend = faces_end();
for (f = faces_begin(); f!= fend; ++f) {
   Pitem[f] = FP.make_block(f);
   clear_face_cycle_entries(f);
}
```

Now we take care of the simplification critereon (2.) of page 123. We only iterate halfedge pairs (uedges). When the marks of the incident faces agree with the mark of the uedge, we union the items of the faces if they are different. Special treatment is required for incident vertices if they become isolated when their last icident uedge is deleted.

⟨*simplify via non-separating halfedges*⟩≡
```
Halfedge_iterator e = halfedges_begin(), en,
                  eend = halfedges_end();
for(; en=e, ++(++en), e != eend; e=en) {
  if ( keep(e) ) continue;
  if ( mark(e) == mark(face(e)) &&
       mark(e) == mark(face(twin(e))) ) {
    if ( !FP.same_block(Pitem[face(e)],
                        Pitem[face(twin(e))]) ) {
      FP.union_blocks( Pitem[face(e)],
                       Pitem[face(twin(e))] );
    }
    if ( is_closed_at_source(e) )       set_face(source(e),face(e));
    if ( is_closed_at_source(twin(e)) ) set_face(target(e),face(e));
    delete_halfedge_pair(e);
  }
}
```

Now we recollect all face cycles and assign them to the face object $f$ that refers to the partition item obtained by a find operation. In each face cycle we determine the halfedge *e_min* whose target has a minimal embedding (with respect to the lexicographic order on points). If *e_min* and $next(e\_min)$ form a left turn they are part of an outer face cycle, otherwise of a hole face cycle. We associate all edges in the face cycle with $f$.

⟨*recollect face cycles per blocks*⟩≡
```
CGAL::Unique_hash_map<Halfedge_handle,bool> linked(false);
for (e = halfedges_begin(); e != eend; ++e) {
  if ( linked[e] ) continue;
  Halfedge_around_face_circulator hfc(e),hend(hfc);
  Halfedge_handle e_min = e;
  Face_handle f = FP.inf(FP.find(Pitem[face(e)]));
  CGAL_For_all(hfc,hend) {
    set_face(hfc,f);
    if ( K.compare_xy(point(target(hfc)), point(target(e_min))) < 0 )
      e_min = hfc;
    linked[hfc]=true;
  }
  Point p1 = point(source(e_min)),
        p2 = point(target(e_min)),
        p3 = point(target(next(e_min)));
  if ( K.orientation(p1,p2,p3) > 0 ) set_halfedge(f,e_min); // outer
  else set_hole(f,e_min); // store as inner
}
```

After the previous simplification we still have to take care of the vertex related simplifications (1.) and (3.). In case that a vertex has outdegree two, that the two incident edges are embedded collinearly, and that all three objects have the same mark, we remove the vertex by joining the two uedges into one. In case that a vertex is isolated and its mark agrees with the incident face we remove the vertex. Otherwise, we anchor the vertex in the face by adding it to the isolated vertex list. Note that the face link of each isolated vertex was either already set in the face creation phase, or in the chunk ⟨*simplify via non-separating halfedges*⟩ when the last incident edge was deleted.

⟨*simplify via vertices*⟩≡
```
Vertex_iterator v, vn, vend = vertices_end();
for(v = vertices_begin(); v != vend; v=vn) {
  vn=v; ++vn;
  if ( is_isolated(v) ) {
    if ( mark(v) == mark(face(v)) ) delete_vertex_only(v);
    else set_isolated_vertex(face(v),v);
  } else { // v not isolated
    Halfedge_handle e2 = first_out_edge(v), e1 = previous(e2);
    Point p1 = point(source(e1)), p2 = point(v),
          p3 = point(target(e2));
    if ( has_outdeg_two(v) &&
         mark(v) == mark(e1) && mark(v) == mark(e2) &&
         (K.orientation(p1,p2,p3) == 0) )
      merge_halfedge_pairs_at_target(e1);
  }
}
```

Finally we discard all face objects that have been victims of unification but do not represent the unified face.

⟨*remove superfluous face objects*⟩≡
```
Face_iterator fn;
for (f = faces_begin(); f != fend; f=fn) {
  fn=f; ++fn;
  partition_item pit = Pitem[f];
  if ( FP.find(pit) != pit ) delete_face(f);
}
```

The following analysis of the partition data structure is due to Tarjan [Tar83].

**Fact 3:** A sequence of $m$ union and find operations starting from $n$ singleton blocks can be done in time $O(m\alpha(m,n))$ with a partition data structure that is based on union by rank and path compression. In this time bound $\alpha$ is the very slowly growing inverse of a suitably defined Ackermann function.

We can therefore summarize the runtime of the simplification action.

**Lemma 4.7.6:** Assume that $P$ is a plane map with the properties cited in the introduction of this section. Then the method *simplify*( ) runs in time $O(n\,\alpha(kn,n))$ where $n$ is the size of $P$, $kn$ is a bound for the number of face unifications and find operations, and $\alpha$ is the function mentioned above.

*Proof.* The number of edges and faces of $P$ is linear in $n$. The number of union operations is bounded by the number of faces, and the number of find operations is bounded by three times[14] the number of edges plus the number of faces. □

Note that after the simplification the plane map output has again the input properties of the overlay calculation operation from Section 4.7.4.

## 4.8 A Generic Segment Sweep Framework

This document describes a generic sweep algorithm of line segments along the lines of the algorithm which is part of the LEDA library. We basically transferred the segment sweep algorithm as described in the LEDA book [MN99] into our generic sweep framework *generic_sweep*. We descibe special adaptations and refer the user to the description in [MN99, chapter 10] for a deeper understanding.

Calculating the overlay of a set of segments is not anymore a theoretical problem. We know its complexity and there are many existing presentations of it. We could have described this module as a black box and just stated the input and output properties of it. We decided to add the implementation description. This presentation stresses the techniques of generic programming and completes the layered design of Nef polyhedra from the interface class down to the sweep engine. Users who have the corresponding insights can just skip the whole section.

To use our generic sweep framework we implement a traits model *Segment_overlay_traits* which is plugged into the class *generic_sweep*<*T*> (the bottom layer). For the concept of the parameter *T* please refer to class *GenericSweepTraits* in the appendix on page 197. For the functionality of class *generic_sweep* see the manual page on page 195. *generic_sweep* < *Segment_overlay_traits* < ... ≫ is a generic sweep framework for the calculation of the overlay of segments.

[14] look for the *find*( ) and *same_block*( ) operations above. The latter uses two find operations.

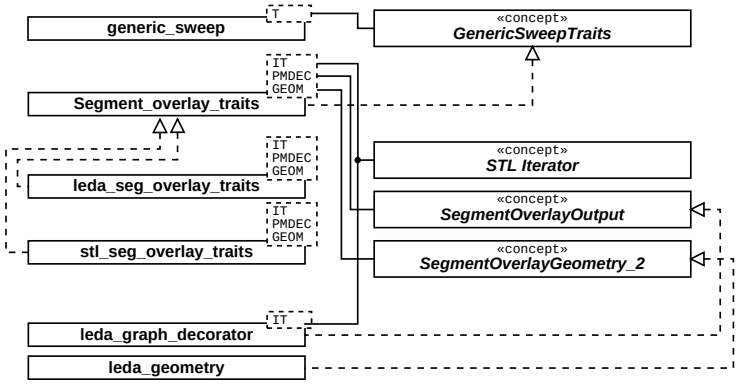



Figure 4.14: The design of the segment overlay module. *Segment_overlay_traits* implements the concept *GenericSweepTraits*. There are actually two instances *leda_seg_overlay_traits* and *stl_seg_overlay_traits* realizing *Segment_overlay_traits* depending on the module configuration. The three template parameters allow an adaptation depending on input, output, and geometry.

If you browse the original algorithm *SWEEP_SEGMENTS* with respect to code dependencies, you find that it is hard-wired to several LEDA modules. The wires of the original algorithm are the *geometric kernel* which is used, the *input interface* which is a list of segments, and the *output interface* which is a LEDA embedded graph. We decouple the above from concrete data types by introducing concepts for the three units: input, output, geometry.

The *input concept* is easy. We use iterators defining an iterator range of segments (corresponding to the list of segments in the LEDA sweep).

The *output concept* is a plane map data structure like LEDA plane maps (bidirected, embedded graphs). For an introduction refer to [MN99, chapter 8]. But of course there are several other standard data structures in the literature like the *Halfedge Data Structures* (HDS), and *Directed Cyclic Edge Lists* (DCEL). See for example the textbooks [dBvKOS97, PS85] and the CGAL HDS implementation in the manual [CGA]. Sometimes the output has to be enriched by some additional bookkeeping data structures (e.g. maps) to associate additional information to the vertices and edges of the graph.

The *geometric concept* contains the geometry used for the algorithmic decisions of the sweep: geometric types like points and segments and the primitive operations on them. Our correctness considerations are based on affine planar geometry. However this sweep framework works also instantiated with other geometry models.

The genericity is achieved by encapsulating the three concepts into three template parameters of *Segment_overlay_traits*<> which allows a user to transport his geometry, geometric primitives and his input and output structure into the segment sweep framework. We will describe the concept of the geometric types, primitives and the concept of the output graph structure below. We first give an abstract introduction of the output produced.

The ouput of the algorithm is the result of transformations of an output object triggered by method calls of the traits class. There are some methods which can be used to manipulate a graph structure $G = (V, E)$. If the implementation of those graph manipulation methods follows the semantic description below then the output graph obtains a certain structure. Additionally there is also some kind of message passing associated with the output structure. This allows a user to refine necessary additional information from the sweep.

Assume the output contains a graph structure $G = (V,E)$ which can represent an embedded plane map (a bidirected graph where reversal edges are paired and where the nodes are embedded into the plane by associating point coordinates). We state some properties of the output production:

O1. All end points and intersection points of segments are called events and trigger calls to create new nodes $v \in V$ which obtain the knowledge about their embedding via a point $p$. The creation is done in the lexicographic order on points as specified by the user in the geometric traits class.

O2. The sweep explores the skeleton of the planar subdivision induced by the set of input segments of the iterator range. Thus for each segment $s$ there are method calls which can be used to create edges in $G$ such that the straight line embedding of their union corresponds to $s$.

O3. Each halfedge $e \in E$ gets to know a list of input segments supporting its straight line embedding.

O4. Each node $v \in V$ gets to know a halfedge below (vertical ray-shooting property) where degeneracies are broken with a left-closed perturbation scheme (all edges include their left source node during the ray shoot). Additionally each node $v \in V$ gets to know the input segments which start at, end at or contain its embedding $point(v)$. Finally each node gets to know explicitly if it is supported by a trivial input segment.

O5. If the edge creation operations follow the semantic description then each node $v$ has an adjacency list such that visiting all adjacent nodes while iterating the adjacency list corresponds to a counterclockwise rotation around $v$ (the straight line drawing of $G$ is counterclockwise *order-preserving*). All adjacency lists additionally have the property of a *forward-prefix*. This means that forward-oriented edges[15] build a prefix in each adjacency list.

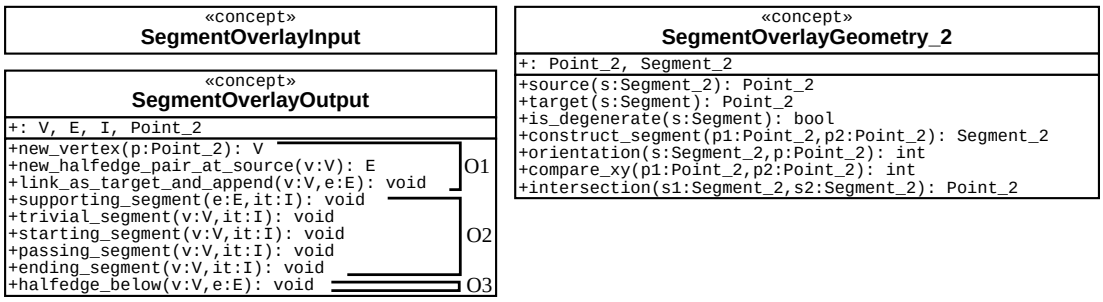



Figure 4.15: The three concepts that allow adaptation of the generic segment sweep. In the output concept the abbreviations are *V* for *Vertex_handle*, *E* for *Halfedge_handle*, and *I* for *Iterator*.

The interfaces of the three concepts are depicted in figure 4.15. For the actual semantics please consult the manual pages for *SegmentOverlayOutput* on page 193 and *SegmentOverlayGeometry_2* on page 194 in the appendix.

### 4.8.1 Formalizing the sweep — Invariants

We sweep the plane from left to right by a vertical line *SL*. Whenever we encounter an event point we have to take actions to produce the output structure. Both endpoints of each segment and non-degenerate intersection points of any two non-overlapping segments define our *events*. To ensure the

[15] an edge $e$ is called forward-oriented when $point(source(e)) <_{lex} point(target(e))$.

correct actions we resort to a list of invariants on which we can rely just before we encounter an event and which we ensure by certain actions for the status thereafter.

The sweep is determined by an interaction between two major data structures, an event queue *XS* and a sweep status structure *YS*. The event queue *XS* controls the stepping of *SL* across the plane. We store a point as the key of each event. The order of the events corresponds to the lexicographical order on all points as defined in the geometry kernel. *YS* stores segments intersecting the sweep line *SL* ordered according to their intersection points from bottom to top. Note that for a segment *s* in *YS* $source(s) <_{lex} target(s)$ according to our lexicographic order on points.

The above description talks about vertical sweep lines and geometry which is left and right of the sweep line. As soon as there are events with identical *x*-coordinates and vertical segments we have to be more accurate. Imagine a sweep line which is slanted by an infinitesimal angle counterclockwise thus processing the points on the vertical line bottom-up. A more accurate intuition is created by a sweepline consisting of an infinitesimal small step down, where the vertical ray down is right of the current sweep position and the vertical ray up is left of the current sweep position and the horizontal step marks the current position while going from $-\infty$ to $+\infty$ along the *y*-axis of the coordinate system. For an extensive treatment please refer to [MN99, chapter 10].

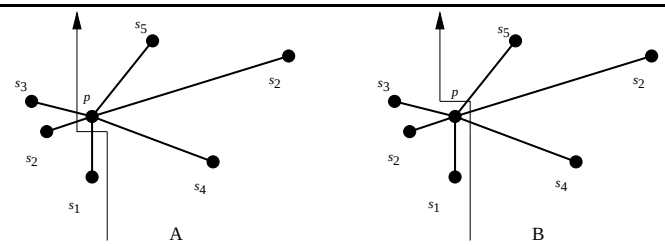



Figure 4.16: Sketching the sweepline intuition in the case of degeneracies. The left figure shows the sweepline before the event at *p*. The right figure shows the figure just after the event at *p*.

We treat the degenerate cases of several segments ending, intersecting, and starting in one point explicitly in our code. We also handle the possibility that several segments overlap. This implies that just before an event the sweep line may intersect a whole bundle of segments containing the event. Some extend through it, some end there. And just after the event the bundle of the segments extending though the event may be enriched by several segments starting at the event. The segments of both bundles can be ordered according to their points of intersection with the sweep line. In case of overlapping segments their point of intersection with the sweep line is identical. To break the tie we take the order on the identity of each input segment[16]. Note that due to the degeneracy events can have multiple character.

**Invariant 1:** The event queue *XS* is a sorted sequence of items <*p*,*x*> called events, where *p* is the embedding point of the event and *x* is an associated information link. *Starting* and *ending* events refer to the endpoints of all segments. *Intersection* events are defined by the points of intersection of two non-overlapping segments. At any sweep position *XS* contains all starting and ending events and all intersection events right of *SL* that are the result of an intersection of segments that are neighbors in *YS*. The order in *XS* is the lexicographic order on points *compare_xy*( ) introduced by the geometry concept.

[16]Internally we handle segments via pointers. Then their identity is just the memory address.

**Invariant 2:** The sweep status structure *YS* is a sorted sequence of items <*s*,*y*> where *s* is a segment intersected by the sweep line. The order is defined by the points of intersection of the segments and the sweep line from bottom to top.

Consider any bundle of segments ending at or extending through an event. We want to save on geometric calculations. Therefore the information slot of the items in *YS* is used to identify such bundles.

**Invariant 3:** Let <*s*,*y*> and <*s*',*y*'> be two successing items in *YS*. The information *y* is used as a flag how the two segments are geometrically related. If *s* and *s*' are overlapping at the current sweep position then *y* points to its successor item <*s*',*y*'> in *YS*. If *s* and *s*' intersect right of the sweep line then *y* points to the corresponding event in *XS*. Otherwise it is the null handle.

Additionally for all items *it* we know an edge in the output graph that is supported by the segment via a map $Edge\_of[it]$.

**Invariant 4:** For all intersection events and ending events <*p*,*x*> in *XS* the information *x* is a link to an item <*s*,*y*> in *YS* such that the segment *s* contains *p*. For ending events the invariant is established as soon as the segment *s* enters *YS*.

This construction allows us to shortcut from an event into the range of interest within *YS* without using the standard (logarithmic) search operations of *YS*.

**Lemma 4.8.1:** Starting from any intersection or ending event we can identify the bundle of segments/items in *YS* ending at or extending through the event in time proportional to the size of the bundle.

*Proof.* Consider an event <*p*,*x*> that is not only starting event. Then *x* is a link to an item <*s*,*y*> in *YS* such that *s* contains *p* due to Invariant 4. Iterating locally up and down starting from *x* allows us to identify the range of items in *YS* whose segments contain *p* by the marking links according to Invariant 3. □

At last, at each event we ensure partial output correctness that leads to global output correctness after the last event.

**Invariant 5:** Assume the output model's operations are defined according to our specification. Then the overlay of all segments that are fully left of our sweep line is correctly calculated.

The following hashing idea saves again on geometric calculations and search.

**Invariant 6:** For each pair of segments $(s1,s2)$ in *YS* and intersecting right of *SL* which have been neighbors in *YS* once (and might have been separated afterwards) there's a hash table short-cut to the corresponding intersection event $it = IEvent(s1,s2)$.

Note that the algorithmic correctness of this framework has two aspects. There are global considerations and local considerations. The global considerations concern issues like why the algorithm terminates and why it does calculate the output we ask for. The local considerations concern the aspects of the event handling. Namely, why our event handling and the intialization phase of our framework ensures the invariants stated above. Taking both issues together we obtain the certainty that our code module actually calculates the overlay correctly.

For the global correctness we will see that all starting and ending events are handled in our code and inflated into the machinery in the initialization phase. One exception that we have to incorporate

into our code is the occurance of trivial segments. The final question is if we catch all intersection events? There's a trivial observation why we cannot miss any such event.

**Lemma 4.8.2:** If we ensure that all our invariants hold at all events then we cannot miss any intersection event.

*Proof.* Assume we miss an intersection event *ev*. If we miss several take the lexicographically smallest one. Then the event just before *ev* was correctly treated and the two segments that imply *ev* are neighbors in *YS*. But Invariant 1 tells us that *ev* is in *XS* which leads to a contradiction. □

Note that global termination is not a big issue in sweep frameworks. As soon as all events have gone we stop the iteration. Note finally that if we locally keep Invariant 5 just after each event then we also know that our result is correctly calculated. We now will link the above insights to the code.

### 4.8.2 Two generic sweep traits models

We use our generic plane sweep paradigm to execute the sweep. We offer two models to plug into the *generic_sweep* framework. For the concept see *GenericSweepTraits* in the appendix. One based on LEDA [MN99] and including several runtime optimizations, the other one based purely on STL data structures [MS96]. This design was necessary to allow using the sweep module when CGAL is installed without the presence of LEDA.

### 4.8.3 The LEDA traits model

Our class obtains three template types which have to be models for the corresponding concepts described below.

⟨*leda segment overlay traits class*⟩≡
```
template <typename IT, typename PMDEC, typename GEOM>
class leda_seg_overlay_traits {
public:
  ⟨leda introducing the types from the traits⟩
  ⟨leda internal segment type⟩
  // types interfacing the generic sweep frame:
  ITERATOR its, ite;
  OUTPUT&  GO;
  const GEOMETRY& K;
  ⟨leda order types for segments and points⟩
  ⟨leda sweep data structures⟩
  ⟨leda helping operations⟩
  ⟨leda operation for keeping the intersection invariant⟩
  ⟨leda initialization of the sweep⟩
  ⟨leda iteration control⟩
  ⟨leda handling the event⟩
  ⟨leda postprocessing of the sweep⟩
}; // leda_seg_overlay_traits
```

The following types are introduced by the traits classes. See the the concept descriptions *SegmentOverlayOutput* for *PMDEC* and *SegmentOverlayGeometry_2* for *GEOM*. The iterator concept required is that of an STL input iterator.

⟨*leda introducing the types from the traits*⟩≡
```
typedef IT                                ITERATOR;
typedef std::pair<IT,IT>                  INPUT;
typedef PMDEC                             OUTPUT;
typedef typename PMDEC::Vertex_handle     Vertex_handle;
typedef typename PMDEC::Halfedge_handle   Halfedge_handle;
typedef GEOM                              GEOMETRY;
typedef typename GEOMETRY::Point_2        Point_2;
typedef typename GEOMETRY::Segment_2      Segment_2;
```

We define an internal segment type *ISegment* based on a pointer to allow two constructions. At first we want to be able to couple the internal segment to the input object. We achieve this by maintaining both as a pair *two_tuple*<*Segment*,*ITERATOR*> in a list. Secondly our internal segment type is just a pointer to such a pair. We can thus not only compare internal segments geometrically by the pair's first component, but also check identity by the pointer address.

⟨*leda internal segment type*⟩≡
```
typedef leda_two_tuple<Segment_2,ITERATOR> seg_pair;
typedef seg_pair*                          ISegment;
typedef leda_list<seg_pair>                IList;
typedef typename IList::iterator           ilist_iterator;
```

The predicates below are solely based on a few geometric kernel predicates. The clever observation is the fact, that the comparison predicate *cmp_segs_at_sweepline* is only called with one segment containing the sweep point. We know that assertion from the specification of LEDA sortseqs. Compared to the LEDA sweep algorithm we add non-geometric sentinel segments to avoid checking of boundary cases.

⟨*leda order types for segments and points*⟩≡
```
class cmp_segs_at_sweepline : public leda_cmp_base<ISegment>
{ const Point_2& p;
  ISegment s_bottom, s_top; // sentinel segments
  const GEOMETRY& K;
public:
 cmp_segs_at_sweepline(const Point_2& pi,
   ISegment s1, ISegment s2, const GEOMETRY& k) :
   p(pi), s_bottom(s1), s_top(s2), K(k) {}
 int operator()(const ISegment& is1, const ISegment& is2) const
 { // Precondition: p is identical to the left endpoint of s1 or s2.
   if ( is2 == s_top || is1 == s_bottom ) return -1;
   if ( is1 == s_top || is2 == s_bottom ) return 1;
   if ( is1 == is2 ) return 0;
   const Segment_2& s1 = is1->first();
   const Segment_2& s2 = is2->first();
   int s = 0;
   if ( p == K.source(s1) )      s =   K.orientation(s2,p);
```

```
    else if ( p == K.source(s2) ) s = - K.orientation(s1,p);
    else ASSERT(0,"compare error in sweep.");
    if ( s || K.is_degenerate(s1) || K.is_degenerate(s2) )
      return s;
    s = K.orientation(s2,K.target(s1));
    if (s==0) return ( is1 - is2 );
    // overlapping segments are not equal
    return s;
  }
};
```

The lexicographic order on points is just transferred from the geometric kernel.

⟨*leda order types for segments and points*⟩+≡

```
struct cmp_pnts_xy : public leda_cmp_base<Point_2>
{ const GEOMETRY& K;
public:
 cmp_pnts_xy(const GEOMETRY& k) : K(k) {}
 int operator()(const Point_2& p1, const Point_2& p2) const
 { return K.compare_xy(p1,p2); }
};
```

We use several LEDA data structures. The x-structure *XS* is an ordered sequence of items based on the key type *Point_2*. For the item concept of LEDA refer to the manual [MNSU99] or the LEDA book. We associate a link into the y-structure for intersection and ending events to save on unnecessary search operations within *YS*. When we reach an event at *p_sweep* we can thus shortcut to find an item in *YS* which contains a segment (as its key) containing *p_sweep*. The y-structure *YS* is a sorted sequence of *ISegments*. The associated *seq_item* link serves two purposes: (1) it bundles segments in *YS* together by pointing to the (lexicographic) next event which they contain. (2) it bundles segments which overlap. See the LEDA book for a more elaborate description.

During the sweep we associate edges from the constructed output graph to the items in YS. We use a hash map *map*<*seq_item*, *Halfedge_handle*> *Edge_of* for this purpose. Finally for each event point there is a possible sequence of input segments starting at the event. To maintain this sequence we use a priority queue *p_queue*<*Point_2*, *ISegment*> *SQ*.

When two segments *s1*, *s2* become neighbors in *YS* we check if they intersect right of the sweep line. If they do we calculate the intersection point *p* and insert a corresponding event into *XS*. Now it can happen that *s1* and *s2* get again separated by a new segment before *p* is reached. We have several possibilities in this case. We could remove the event at *p* again to keep the space bound implied by the Invariant 1. However the size of the output structure anyway comprises the space for keeping the event in *XS*. We can even do better with respect to runtime. Instead of recalculating the geometric intersection information when *s1* and *s2* become neighbors again we can try to recover the previously calculated event from the two dimensional hash *map*2<*ISegment*, *ISegment*, *seq_item*> *IEvent*.

The additional members are used as a central place for data storage. *event* provides a handle on the current event queue item. *p_sweep* is the position of the current event which is also used from the segment comparison object *SLcmp*.

⟨*leda sweep data structures*⟩≡

```
typedef leda_sortseq<Point_2,seq_item>      EventQueue;
typedef leda_sortseq<ISegment,seq_item>     SweepStatus;
typedef leda_p_queue<Point_2,ISegment>      SegQueue;
typedef leda_map<seq_item,Halfedge_handle>  AssocEdgeMap;
typedef leda_slist<ITERATOR>                IsoList;
typedef leda_map<seq_item, IsoList* >       AssocIsoMap;
typedef leda_map2<ISegment,ISegment,seq_item> EventHash;

  seq_item                  event;
  Point_2                   p_sweep;
  cmp_pnts_xy               cmp;
  EventQueue                XS;
  seg_pair                  sl,sh;
  cmp_segs_at_sweepline     SLcmp;
  SweepStatus               YS;
  SegQueue                  SQ;

  EventHash                 IEvent;
  IList                     Internal;
  AssocEdgeMap              Edge_of;
  AssocIsoMap               Isos_of;

  leda_seg_overlay_traits(const INPUT& in, OUTPUT& G,
    const GEOMETRY& k) :
    its(in.first), ite(in.second), GO(G), K(k),
    cmp(K), XS(cmp), SLcmp(p_sweep,&sl,&sh,K), YS(SLcmp), SQ(cmp),
    IEvent(0), Edge_of(0), Isos_of(0) {}
```

We define some code short cuts.

⟨*leda helping operations*⟩+≡

```
Point_2 source(ISegment is) const
{ return K.source(is->first()); }
Point_2 target(ISegment is) const
{ return K.target(is->first()); }
ITERATOR original(ISegment s) const
{ return s->second(); }

int orientation(seq_item sit, const Point_2& p) const
{ return K.orientation(YS.key(sit)->first(),p); }

bool collinear(seq_item sit1, seq_item sit2) const
{ Point_2 ps = source(YS.key(sit2)), pt = target(YS.key(sit2));
  return ( orientation(sit1,ps)==0 &&
           orientation(sit1,pt)==0 );
}
```

Most events trigger changes in the segment sequence along the sweep line. We have to reflect such changes in a test for new intersection events right of the sweep line as soon as two segments become neighbors. The following code ensures the Invariants 2, 3, 4 and uses the hash tuning of Invariant 6. *s1* is the successor of *s0* in *YS*, hence, *s0* and *s1* intersect right or above of the event iff *target*(*s1*) is not left of the line supporting *s0*, and *target*(*s0*) is not right of the line supporting *s1*. In this case we intersect the underlying lines.

⟨*leda operation for keeping the intersection invariant*⟩≡
```
void compute_intersection(seq_item sit0)
{
  seq_item sit1 = YS.succ(sit0);
  if ( sit0 == YS.min_item() || sit1 == YS.max_item() ) return;
  ISegment s0 = YS.key(sit0);
  ISegment s1 = YS.key(sit1);
  int or0 = K.orientation(s0->first(),target(s1));
  int or1 = K.orientation(s1->first(),target(s0));
  if ( or0 <= 0 && or1 >= 0  ) {
    seq_item it = IEvent(YS.key(sit0),YS.key(sit1));
    if ( it==0 ) {
      Point_2 q = K.intersection(s0->first(),s1->first());
      it = XS.insert(q,sit0);
    }
    YS.change_inf(sit0, it);
  }
}
```

**Event Handling**

We start with the knowledge that our invariants from Section 4.8.1 hold. First we create a new vertex *v* in the ouput structure. Then we work in four phases: (1) We handle the ingoing bundle which ends at *v*. (2) We communicate all the knowledge about the new vertex (3) We have to deal with the segments starting at *p_sweep*. (4) We clean up to reestablish missing invariants.

⟨*leda handling the event*⟩≡
```
void process_event()
{

  Vertex_handle v = GO.new_vertex(p_sweep);
  seq_item sit = XS.inf(event);
```
⟨*leda handling ending and passing segments*⟩
⟨*leda completing additional information of the new vertex*⟩
⟨*leda inserting new segments starting at nodes*⟩
⟨*leda enforcing the invariants for YS*⟩
```
}
```

We first have to locate the bundle going through *p_sweep*. We deviate from the implementation of the LEDA algorithm *SWEEP_SEGMENTS* in one respect. For each segment in *YS* we store a bidirected edge pair extending along the segment. When we reach an event point we connect these edges to the newly created node. Note that this change is necessary if you use halfedge data structures for the output. The original approach used temporarily incomplete edge pairs (only forward directed halfedges) and coupled and embedded them in a postprocessing phase. But space minimally maintained halfedges like those of the CGAL HDS can only exist in pairs.

If there is a non-nil item $sit = XS.inf(event)$ associated with *event*, $key(sit)$ is either an ending or passing segment. We use *sit* as an entry point to compute the bundle of segments ending at or passing through *p_sweep*. In particular, we compute the first (*sit_first*) and the successor (*sit_succ*)) and

predecessor (*sit_pred*) items.

⟨*leda handling ending and passing segments*⟩≡
```
seq_item sit_succ(0), sit_pred(0), sit_pred_succ(0), sit_first(0);
if (sit == nil)
  ⟨leda check p_sweep in YS⟩
/* If no segment contains p_sweep then sit_pred and sit_succ are
   correctly set after the above locate operation, if a segment
   contains p_sweep sit_pred and sit_succ are set below when
   determining the bundle.*/
if (sit != nil) { // key(sit) is an ending or passing segment
  ⟨leda determine upper bundle item sit_succ⟩
  ⟨leda hash upper intersection event⟩
  ⟨leda walk ingoing bundle and trigger graph updates⟩
  ⟨leda reverse continuing bundle edges⟩
} // if (sit != nil)
```

As $sit == nil$ we do not know if a segment stored in *YS* does contain *p_sweep*. We have to query *YS* with a trivial segment $(p\_sweep, p\_sweep)$ to find out. Two results are possible. Either a segment referenced hereafter by *sit* constains the event point or we determine the two segments above and below *p_sweep* in *sit_pred* and *sit_succ*.

⟨*leda check p_sweep in YS*⟩≡
```
{
  Segment_2 s_sweep = K.construct_segment(p_sweep,p_sweep);
  seg_pair sp(s_sweep,ITERATOR());
  sit_succ = YS.locate( &sp );
  if ( sit_succ != YS.max_item() &&
       orientation(sit_succ,p_sweep) == 0 )
    sit = sit_succ;
  else  {
    sit_pred = YS.pred(sit_succ);
    sit_pred_succ = sit_succ;
  }
}
```

We first walk up as long as the event is contained in the segment referenced via *sit*.

⟨*leda determine upper bundle item sit_succ*⟩≡
```
while ( YS.inf(sit) == event ||
        YS.inf(sit) == YS.succ(sit) ) // overlapping
  sit = YS.succ(sit);
sit_succ = YS.succ(sit);
seq_item sit_last = sit;
```

We hash the upper event according to Invariant 6.

⟨*leda hash upper intersection event*⟩≡
```
seq_item xit = YS.inf(sit_last);
if (xit) {
  ISegment s1 = YS.key(sit_last);
  ISegment s2 = YS.key(sit_succ);
  IEvent(s1,s2) = xit;
}
```

We walk the ingoing bundle down again and trigger the edge closing calls for all items in the bundle (except overlapping segments). Note that after this code chunk we have: (i) the bundle is empty if $succ(sit\_pred) == sit\_first == sit\_succ$, or (ii) the bundle is not empty if $sit\_first != sit\_succ$.

The actions on the bundle are easy to specify. We have to glue one edge per segment to the newly created node except when two segments overlap. Note that the walk top-down over the bundle implies the order-preserving embedding of the graph. Note also how we pass the messages about the segments supporting the event via the corresponding methods of the output object.

⟨*leda walk ingoing bundle and trigger graph updates*⟩≡
```
bool overlapping;
do {
  ISegment s = YS.key(sit);
  seq_item sit_next = YS.pred(sit);
  overlapping = (YS.inf(sit_next) == sit);
  Halfedge_handle e = Edge_of[sit];
  if ( !overlapping ) {
    GO.link_as_target_and_append(v,e);
  }
  GO.supporting_segment(e,original(s));
  if ( target(s) == p_sweep ) { // ending segment
    if ( overlapping ) YS.change_inf(sit_next,YS.inf(sit));
    YS.del_item(sit);
    GO.ending_segment(v,original(s));
  } else {  // passing segment
    if ( YS.inf(sit) != YS.succ(sit) )
      YS.change_inf(sit, seq_item(0));
    GO.passing_segment(v,original(s));
  }
  sit = sit_next;
}
while ( YS.inf(sit) == event || overlapping ||
        YS.inf(sit) == YS.succ(sit) );
sit_pred = sit;
sit_first = sit_pred_succ = YS.succ(sit_pred); // first item of bundle
```

We have to ensure that segments that continue through the event point have a reversed order within *YS* when *p_sweep* has been passed. This ensures the correct order of *YS* with respect to Invariant 2. Some complication stems from overlapping segments. Their order based on identity may not be changed.

⟨*leda reverse continuing bundle edges*⟩≡
```
while ( sit != sit_succ ) {
  seq_item sub_first = sit;
  seq_item sub_last  = sub_first;
  while (YS.inf(sub_last) == YS.succ(sub_last))
    sub_last = YS.succ(sub_last);
  if (sub_last != sub_first)
    YS.reverse_items(sub_first, sub_last);
  sit = YS.succ(sub_first);
}
// reverse the entire bundle
if (sit_first != sit_succ)
  YS.reverse_items(YS.succ(sit_pred),YS.pred(sit_succ));
```

For the new node *v* we pass some information to the output structure. We post the halfedge below, we post all trivial input segments supporting the node. We obtain that information from the hash structure *Isos_of* filled during the initialization phase.

⟨*leda completing additional information of the new vertex*⟩≡
```
assert(sit_pred);
GO.halfedge_below(v,Edge_of[sit_pred]);
if ( Isos_of[event] != 0 ) {
  const IsoList& IL = *(Isos_of[event]);
  slist_item iso_it;
  for (iso_it = IL.first(); iso_it; iso_it=IL.succ(iso_it) )
    GO.trivial_segment(v,IL[iso_it] );
  delete (Isos_of[event]); // clean up the list
}
```

We insert all segments starting at *p_sweep* into *YS* and create the links within *YS* to mark items with overlapping segments.

⟨*leda inserting new segments starting at nodes*⟩≡
```
ISegment next_seg;
pq_item next_it = SQ.find_min();
while ( next_it &&
        (next_seg = SQ.inf(next_it), p_sweep == source(next_seg)) ) {
  seq_item s_sit = YS.locate_succ(next_seg);
  seq_item p_sit = YS.pred(s_sit);
  if ( YS.max_item() != s_sit &&
       orientation(s_sit, source(next_seg) ) == 0 &&
       orientation(s_sit, target(next_seg) ) == 0 )
    sit = YS.insert_at(s_sit, next_seg, s_sit);
  else
    sit = YS.insert_at(s_sit, next_seg, seq_item(nil));
  assert(YS.succ(sit)==s_sit);
  if ( YS.min_item() != p_sit &&
       orientation(p_sit, source(next_seg) ) == 0 &&
       orientation(p_sit, target(next_seg) ) == 0 )
```

```
    YS.change_inf(p_sit, sit);
  assert(YS.succ(p_sit)==sit);
  XS.insert(target(next_seg), sit);
  GO.starting_segment(v,original(next_seg));
  // delete minimum and assign new minimum to next_seg
  SQ.del_min();
  next_it = SQ.find_min();
}
```

In contrast to the original LEDA segment intersection algorithm *SWEEP_SEGMENTS* we create "semi-open" edges starting at the *event* node and supported by the input segment. The iteration again ensures the correct order-preserving embedding at the currently handled node *v*.

⟨*leda inserting new segments starting at nodes*⟩+≡

```
for( seq_item sitl = YS.pred(sit_succ); sitl != sit_pred;
     sitl = YS.pred(sitl) ) {
  if ( YS.inf(sitl) != YS.succ(sitl) ) { // non-overlapping
    Edge_of[sitl] = GO.new_halfedge_pair_at_source(v);
  } else {
    Edge_of[sitl] = Edge_of[ YS.succ(sitl) ];
  }
}
sit_first = YS.succ(sit_pred);
```

Depending on the outgoing bundle we determine possible intersections between new neighbors; if *sit_pred* is no longer adjacent to its former successor we change its intersection event to 0. Note that the following chunk finishes Invariant 1 with the help of the method *compute_intersection*( ).

⟨*leda enforcing the invariants for YS*⟩≡

```
assert(sit_pred); assert(sit_pred_succ);
seq_item xit = YS.inf(sit_pred);
if ( xit ) {
  ISegment s1 = YS.key(sit_pred);
  ISegment s2 = YS.key(sit_pred_succ);
  IEvent(s1,s2) = xit;
  YS.change_inf(sit_pred, seq_item(0));
}
compute_intersection(sit_pred);
sit = YS.pred(sit_succ);
if (sit != sit_pred)
  compute_intersection(sit);
```

**Initialization**

We realize the propositions of the invariants at the beginning in our sweep initialization phase. We insert all segment endpoints into *XS*, insert sentinels into *YS*, and exploit the fact that insert operations into the X-structure leave previously inserted points unchanged to achieve that any pair of endpoints

$p$ and $q$ with $p == q$ are identical (if the geometric point type supports this). Degenerate segments are stored in a list associated to their events. The knowledge about their existence is transferred to the corresponding output object as soon as it is constructed.

⟨*leda initialization of the sweep*⟩≡
```
void initialize_structures()
{
  ITERATOR it_s;
  for ( it_s=its; it_s != ite; ++it_s ) {
    Segment_2 s = *it_s;
    seq_item it1 = XS.insert( K.source(s), seq_item(nil));
    seq_item it2 = XS.insert( K.target(s), seq_item(nil));
    if (it1 == it2) {
      if ( Isos_of[it1] == 0 ) Isos_of[it1] = new IsoList;
      Isos_of[it1]->push(it_s);
      continue;  // ignore zero-length segments in SQ/YS
    }
    Point_2 p = XS.key(it1);
    Point_2 q = XS.key(it2);
    Segment_2 s1;
    if ( K.compare_xy(p,q) < 0 )
      s1 = K.construct_segment(p,q);
    else
      s1 = K.construct_segment(q,p);
    Internal.append(seg_pair(s1,it_s));
    SQ.insert(K.source(s1),&Internal[Internal.last()]);
  }
  // insert a lower and an upper sentinel segment
  YS.insert(&sl,seq_item(nil));
  YS.insert(&sh,seq_item(nil));
}
```

Note the invariants of the sweep loop. The *event* has to be set before the event action.

⟨*leda iteration control*⟩≡
```
bool event_exists()
{
  if (!XS.empty()) {
    // event is set at end of loop and in init
    event = XS.min();
    p_sweep = XS.key(event);
    return true;
  }
  return false;
}
void procede_to_next_event()
{ XS.del_item(event); }
```

The structure is finished as soon as all events have been treated. As we always created edges in pairs and respected the adjacency list order we have no completion phase as in [MN99, chapter 10].

**Runtime Analysis**

We shortly give a runtime analysis of the above implementation. Assume $U(S)$ to be the embedded (undirected) graph created by a proper[17] instantiation of our sweep framework. Let $n$ be the number of input segments of the input set $S$, $n_v$ be the number of nodes of $U(S)$, $n_e$ be the number of (undirected) edges of $U(S)$. In the presence of overlapping segments let $\bar{n}_e := \sum_{e \text{ edge of } U(S)} s_e$ where $s_e$ is the number of segments in $S$ supporting edge $e$ of $U(S)$. Note that during the sweep the whole graph is constructed and explored. Then obviously all graph related operations like creation, and support messaging take time $O(n_v + \bar{n}_e)$.

The sweep initialization takes time $O(n \log n)$. There are $n_v$ events and at each event the sweep status structures are manipulated a constant number of times. Pass again through the event handling routine. All segments are inserted into $YS$ and deleted from $YS$ once (during the whole sweep) and therefore that cost adds up to $O(n \log |YS|)$. The removal from $SQ$ adds up $O(n \log n)$. Finally all events are inserted and deleted once and this takes $O(|XS| \log |XS|)$. When subsequences of $YS$ are explored and swapped, this cost can again be dedicated to the exploration of the graph and is therefore subsumed in the $O(n_v + \bar{n}_e)$ from above.

We have to bound the size of $XS$ and $YS$. Natural bounds are $|XS| = O(n_v)$ (the number of nodes) and $|YS| = O(n)$ (the number of segments). Therefore accumulating all of the above we obtain $O(n_v + \bar{n}_e + n \log n + n_v \log(n + n_v)) = O(n_v + \bar{n}_e + (n + n_v) \log(n + n_v))$. Note that $\bar{n}_e$ and $n_v$ can be quadratic in $n$.

**Lemma 4.8.3:** Assume that $n$, $n_v$, $\bar{n}_e$ are defined as above then the runtime of the sweep algorithm is

$$O(n_v + \bar{n}_e + (n + n_v) \log(n + n_v)).$$

### 4.8.4 The STL traits model

We will not show the STL model. It is highly analogously, sometimes simpler, but also a less efficient due to limitations in the interface of the STL data structures. For the complete implementation see the report [See01].

## 4.9 Conclusions

In this chapter we have designed a software module for planar Nefpolyhedra. We have presented the relevant theory and derived a data type *Nef_polyhedron_2* with a rich interface that fits the needs of many users who approach a software library like CGAL for problem solutions in the domain of planar polyhedral sets. The software module including the interface documentation is part of the CGAL basic library.

The design is clearly modularized and based on generic traits class techniques that allow flexibility without loosing efficiency. The presentation is a stepwise conversion from theory to the implementation facets of the programming project. The shown code parts are expressive and compact. The relevant algorithmic modules meet the optimal runtime bounds. By interweaving code and correctness considerations in literate programming style we hopefully could convince our readers of the correctness of our programs.

We have shown that our approach to a dynamic frame offers one solution to the problem of non-compact structures. Its main advantage is the transparent handling of geometric configurations that

[17] The output methods have to be constant time operations of the correct semantics.

| #lines | op | #V | #E | #F | simple | filtered | rgpn | rgp |
|---|---|---|---|---|---|---|---|---|
| 10 | $\oplus$ | 67 | 121 | 56 | 0.64 | 0.065 | 0.075 | 0.125 |
| 10 | $\cap$ | 162 | 217 | 61 | 0.74 | 0.09 | 0.08 | 0.93 |
| 10 | $\cup$ | 160 | 213 | 60 | 0.75 | 0.09 | 0.07 | 0.92 |
| 20 | $\oplus$ | 230 | 437 | 209 | 2.31 | 0.26 | 0.285 | 2.3 |
| 20 | $\cap$ | 622 | 831 | 241 | 2.89 | 0.36 | 0.3 | 19.2 |
| 20 | $\cup$ | 577 | 744 | 205 | 2.93 | 0.36 | 0.3 | 19.6 |
| 30 | $\oplus$ | 499 | 964 | 467 | 4.01 | 0.48 | 0.575 | -1 |
| 30 | $\cap$ | 1375 | 1821 | 514 | 7 | 0.92 | 0.8 | -1 |
| 30 | $\cup$ | 1410 | 1894 | 540 | 7.55 | 0.9 | 0.76 | -1 |
| 40 | $\oplus$ | 862 | 1680 | 820 | 8.7 | 1.05 | 1.25 | -1 |
| 40 | $\cap$ | 2395 | 3171 | 900 | 12.8 | 1.67 | 1.43 | -1 |
| 40 | $\cup$ | 2509 | 3370 | 960 | 12.9 | 1.71 | 1.42 | -1 |
| 50 | $\oplus$ | 1326 | 2600 | 1275 | 12.1 | 1.54 | 1.83 | -1 |
| 50 | $\cap$ | 3828 | 5111 | 1470 | 21.1 | 2.88 | 2.41 | -1 |
| 50 | $\cup$ | 3809 | 5073 | 1455 | 20.7 | 2.85 | 2.35 | -1 |
| 100 | $\oplus$ | 5149 | 10195 | 5048 | 49.8 | 6.76 | 8.29 | -1 |
| 100 | $\cap$ | 15188 | 20296 | 5826 | 92.6 | 13.5 | 13.1 | -1 |
| 100 | $\cup$ | 15088 | 20073 | 5731 | 92.4 | 13.6 | 13.4 | -1 |
| 150 | $\oplus$ | 11476 | 22799 | 11325 | 146 | 21.2 | 24.2 | -1 |
| 150 | $\cap$ | 34223 | 45801 | 13214 | 217 | 33.7 | 39.8 | -1 |
| 150 | $\cup$ | 33717 | 44785 | 12881 | 217 | 34.1 | 40.2 | -1 |
| 200 | $\oplus$ | 20302 | 40401 | 20100 | 207 | 33.8 | 40 | -1 |
| 200 | $\cap$ | 60043 | 79905 | 22909 | 415 | 66 | 101 | -1 |
| 200 | $\cup$ | 60551 | 80886 | 23352 | 410 | 67.1 | 100 | -1 |

Table 4.1: Running times in seconds on a SUN Ultra-Enterprise-10000 with 333 MHz Ultra SPARC processors.

appear in the standard affine plane but also on the frame when introducing extended points. Available program code for standard affine problems can be easily used as long as the used geometric predicates are extensible to extended points and extended segments. The application of our infimaximal framework separates the geometric issues of the frame addition from the control structure of the implemented algorithm. On the other side our runtime results in the following section show that infimaximal frames can be used without introducing a relevant runtime overhead compared to standard affine geometry.

### 4.9.1 Efficiency

We present runtime results. We performed the following experiments. Starting from $n$ random half-spaces (the coefficients of the boundary line are random integers in $[-n,n]$ and one of the half-spaces is chosen at random) we built a Nef polyhedron by symmetric difference operations. We put the half-spaces at the leaves of a balanced binary tree and formed the symmetric difference of the two children at each internal node. In the root we obtained a Nef polyhedron $P$ of complexity $\Theta(n^2)$. The polyhedron is essentially the arrangement defined by the boundary lines; a vertex, edge, face of the

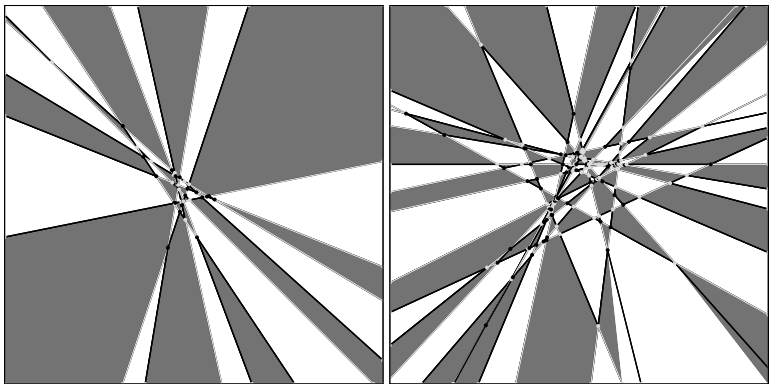


Figure 4.17: Two polyhedra as a result of the exor combination for $n = 10$ and $n = 20$.

arrangement belongs to $P$ iff it is contained in an odd number of the half-spaces[18]. We then took two Nef polyhedra obtained in this way and formed their *intersection* and *union*. We performed this experiment for different values of $n$ ranging from 10 to 200. Table 4.1 shows the results. The line marked $\oplus$ presents the complexity and times for the recursive synthesis of the Nef polyhedron. The columns labeled #V, #E, and #F give the actual number of nodes, edges and faces of the resulting arrangement (deviation from the formulas above are due to degeneracies). The lines labeled $\cap$ and $\cup$ present the results of the corresponding binary operation.

We show the running times for four different implementations. The first two columns show the running times of our second and third implementation of epoints and esegments. The column (marked as *simple*) uses CGAL's homogeneous kernel instantiated with the ring type *RPolynomial* which in turn uses LEDA's multiprecision integer arithmetic. Column *filtered* shows the running times for version three which uses filtered predicates as described in Section 3.4. A comparison of the first two columns shows that our third implementation is much superior to the second.

We also wanted a comparison with the approach based on a concrete frame. LEDA offers a type *rat_gen_polygon* which is closed under regularized boolean operations (a regularized boolean operation discards isolated lower dimensional features by replacing the true result of a boolean operation by the closure of the interior of the result) and allows only a single unbounded face. Regularized boolean operations lead to a simpler topological structure than general boolean operations. We enclosed our half-spaces into a concrete geometric frame (whose size we inferred from the computation of the Nef polygon) and then executed the same set of operation. *Rat_gen_polygons* also use LEDA's sweep for the overlay of two maps. The last column (labeled *rpg*) shows the running times of *rat_polygons*. A $-1$ indicates that the experiment was not run due to the excessively large running times. The excessively bad behavior of *rat_gen_polygon* surprised us. We traced it to the fact that LEDA does not normalize point representations automatically. We added a normalization step after each binary operation. The resulting running times are shown in column *rpgn*.

A comparison of columns *filtered* and *rpg* shows that the filtered implementation is slightly faster than LEDA's *rat_gen_polygons* for the synthesis step and slightly slower for the union and intersection. For $n = 200$ we are faster for all three operations. Our explanation is as follows (we admit that we are not completely sure whether our explanation is the right one): the use of a concrete geometric frame forces us to use large coordinate values for the frame points. The coefficients of our polynomials are

[18]The exact number (assuming non-degeneracy and taking the objects on the frame into account) of vertices, edges, and faces is $n(n-1)/2+2n+4$, $n(n+1)+2n+4$, and $n(n+1)/2+2$, respectively.

smaller (much smaller in the early synthesis steps). For $n = 200$ the coordinate values in the geometric frame become so large that the filter in LEDA's rational geometry kernel start to loose its effectiveness. The filter in epoints stays effective.

It is also interesting to compare columns *simple* and *rpg*. For $n = 10$, *rpg* is superior, for $n = 20$, the two implementations are about the same, and for larger $n$, *simple* wins by a large margin due to the fact that we reduce the polynomial coordinate representation of simple epoints on construction by a polynomial division. Remember that the *rpg* code does not use any reduction.

### 4.9.2 Further Applications and Future Work

We give some more applications of the results of our work. We have successfully used the generic segment overlay sweep module of Section 4.8 in two further application scenarios. One is the creation of a facet structure embedded into a planar subspace in dimension three. Such an application is easy to realize when the geometric kernel uses projection methods for the algorithmic predicates. Another application is the overlay of two maps embedded into the surface of a three-dimensional sphere. Let us call them sphere maps for short. Binary operations of sphere maps can be implemented similar to those in Section 4.7. This problem is a central task when considering the realization of spatial Nef polyhedra. To transfer the insights of Section 4.7 one has to provide solutions according to the following items:

**spherical geometry** — use a spherical geometry kernel consisting of sphere points, sphere segments (parts of large circles[19]), and large circles. To avoid robustness problems, use a kernel similar to that developed by J. Schwerdt [Sch01], who uses predicates of the LEDA three-dimensional rational geometry kernel to implement spherical predicates and constructions.

**sphere map data type** — the main difference in the data type is the absence of unbounded faces. Boundary cycles cannot be separated into outer ones and holes. On the other hand there are degenerate edge structures called loops that have no start and end vertices but represent large circles.

**algorithmic idea** — we divide the whole sphere along its equator into two identical subproblems. The spherical input objects are partitioned by the equator. The sweep line is a half-circle fixed at two oppositely fixed points that are part of the equator. We instantiate our generic segment overlay sweep with a spherical geometry kernel and an output model maintaining and extending the output sphere map. We accumulate face cycle link information as in the planar case[20] to allow the creation of face objects and the accumulation of support information per half-sphere. Finally we stitch the two structures together along the equator and simplify the whole structure to obtain maximal objects.

Binary operations of spatial Nef polyhedra can be again structured into the phases: subdivision - selection - simplification. To store Nef polyhedra we can use a spatial cellular subdivision where the cells are the connected components of Nef faces (a generalization of the plane map for dimension 3. A first design was presented in the research report [DMY93]. As Bieri has shown, it is sufficient to determine the local pyramids of all faces of minimal dimension, to describe the whole Nef complex. A rough outline of binary operations in space could therefore be as follows. Use an infimaximal cube as an extension of our scheme from chapter 3 for symbolic compactification of space. Then vertices

[19] The intersections of planes through the center with the surface of the sphere.

[20] c.f. halfedge_below concept in Section 4.7.

are again the minimal faces and their local view does sufficiently describe the whole topological structure. Use again a scheme as advocated by SGCs for binary operations. The vertex candidates of the result of any binary operation are the vertices of the input structures plus the vertices in the non-degenerate intersection of edge-edge pairs and edge-facet pairs. Use a suitable qualification method (point location) to obtain the local view of the vertices of the output structure with respect to the two input structures. Store the local views as sphere maps. Use the spherical overlayer in the calculation of the local views of the vertices in binary operations. Finally use a synthesis phase for the construction of a spatial structure that is a cellular subdivision of space. A simplification of the cell complex generalizing the methods of Section 4.7 produces again a minimal representation that maintains the connected components of Nef faces.

# Bibliography


[Ame97] N. Amenta. Computational geometry software. In *Handbook of Discrete and Computational Geometry*, pages 951–960. CRC Press, 1997.

[Ant98] H. Anton. *Lineare Algebra*. Spektrum Akademischer Verlag, Berlin, 1998.

[Ash99] O. Ashoff. Algorithmen der modularen Arithmetik. Master's thesis, Universität des Saarlandes, 1999.

[BBP98] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[BDH96] C. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull program for convex hulls. *ACM Transactions on Mathematical Software*, 22:469–483, 1996. havenot.

[BFS98] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. In *Proceedings of the 14th Annual Symposium on Computational Geometry (SCG'98)*, pages 175–183, 1998.

[Bie94] H. Bieri. Boolean and topological operations for Nef polyhedra. In *CSG 94 Set-theoretic Solid Modelling: Techniques and Applications*, pages 35–53. Information Geometers, 1994.

[Bie95] H. Bieri. Nef Polyhedra: A Brief Introduction. *Computing Suppl. Springer-Verlag*, 10:43–60, 1995.

[Bie96] H. Bieri. Two basic operations for Nef polyhedra. In *CSG 96 Set-theoretic Solid Modelling: Techniques and Applications*, pages 337–356. Information Geometers, 1996.

[Bie98] H. Bieri. Representation conversions for Nef Polyhedra. *Computing Suppl. Springer-Verlag*, 13:27–38, 1998.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Annual ACM Symp. on Theory of Computing*, pages 73–83, 1990.

[BMS94] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. SODA 94*, pages 16–23, 1994.

[BN88] H. Bieri and W. Nef. Elementary set operations with d-dimensional polyhedra. In Hartmut Noltemeier, editor, *Computational Geometry and its Applications*, volume 333 of *LNCS*, pages 97–112. Springer, March 1988.

[CDR92] J. Canny, B.R. Donald, and E.K. Ressler. A rational rotation method for robust geometric algorithms. In *Proceedings of the 8th Annual Symposium on Computational Geometry (SCG '92)*, pages 251–260, 1992.

[CGA] The CGAL home page. www.cgal.org.

[CGG+91a] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language reference manual*. Springer, 1991.

[CGG+91b] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V : language reference manual*. Springer, / 1st ed. edition, 1992/1991.

[CMS93] K.L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. In *Computational geometry : theory and applications*, volume 3, pages 185–212, Amsterdam, 1993. Elsevier.

[Coh93] H. Cohen. *A Course in Computational Algebraic Number Theory*, volume Graduate Texts in Mathematics 138. Springer Verlag, New York Berlin Heidelberg, 1993.

[com98] ISO/IEC committee. *Programming languages - C++ = Langages de programmation - C++ : international standard ISO/IEC 14882*, volume ISO/IEC 14882:1998(E). ANSI, 1. ed. edition, 1998.

[Cox87] H.S.M. Coxeter. *Projective geometry*. Springer, 2nd ed. 1974. corr. 2nd print. 1994 edition, 1987.

[dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Springer, Berlin, 1997.

[Die97] R. Diestel. *Graph theory*, volume 173 of *Graduate texts in mathematics*. Springer, 1997.

[DMY93] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. Technical Report 2023, INRIA, 1993.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer, 1987.

[Edm67] J. Edmonds. Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards*, 71(B):241–245, 1967.

[EM90] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, January 1990.

[Fer95] V. Ferrucci. Representing Nef Polyhedra. In Kunii Shin, editor, *Proceedings of the 3rd Pacific conference on computer graphics and applications - Pacific Graphics 95*, pages 459–511. World Scientific, 1995.

[FGK+96] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel : a basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 191–202. Springer LNCS 1148, 1996.

[FvW96] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15:223–248, 1996. preliminary version in ACM Conference on Computational Geometry 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[Grü67] B. Grünbaum. *Convex Polytopes*, volume 16 of *Pure and Applied Mathematics : A Series of Monographs and Textbooks*. Interscience Publ., London;New York;Sydney, 1967.

[HHK+01a] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proceedings of the 5th Workshop of Algorithmic Engineering*, 2001. to appear.

[HHK+01b] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. Technical Report to appear, Max-Planck-Institut für Informatik, Saarbrücken, 2001.

[Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *CGTA: Computational Geometry: Theory and Applications*, 13, 1999.

[KLN91] M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[Knu98] D.E. Knuth. *The Art of Computer Programming*, volume Volume 2, Seminumerical Algorithms, 3rd edition. Addison-Wesley, 1998.

[LEDa] The LEDA home page. www.mpi-sb.mpg.de/LEDA.

[LEDb] An extended segment sweep package. www.mpi-sb.mpg.de/LEDA.

[Lee96] D. T. Lee. Visualizing geometric algorithms – state of the art. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, volume 1148 of *LNCS*, pages 45–50. Springer-Verlag, 1996.

[Lef71] S. Lefschetz. *Introduction to topology*, volume 11 of *Princeton mathematical series*. Princeton Univ. Press, 8. print. edition, 1971.

[Meh96] Kurt Mehlhorn. Position paper for panel discussion. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, volume 1148 of *LNCS*, pages 51–52. Springer-Verlag, 1996.

[MMN+96] K. Mehlhorn, M. Müller, S. Näher, S. Schirra, M. Seel, C. Uhrig, and J. Ziegler. A computational basis for higher-dimensional computational geometry and applications. Technical Report MPI-I-96-1-016, Max-Planck-Institut für Informatik, Saarbrücken, 1996.

[MMN+97] K. Mehlhorn, M. Müller, S. Näher, S. Schirra, M. Seeland C. Uhrig, and J. Ziegler. A computational basis for higher-dimensional computational geometry and applications. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 254–263, Nice, France, 1997. Association of Computing Machinery (ACM), ACM Press.

[MMS94] J.S.B. Mitchell, D. Mount, and S. Suri. Query-sensitive ray shooting. In *Proceedings of the 10th Annual Symposium on Computational Geometry*, pages 359–368, Stony Brook, NY, USA, June 1994. ACM Press.

[MN94] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[MN99] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MNS+96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking Geometric Programs or Verification of Geometric Structures. In *Proc. of the 12th Annual Symposium on Computational Geometry*, pages 159–165, 1996.

[MNS+99] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *CGTA: Computational Geometry: Theory and Applications*, 12, 1999.

[MNSU99] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual*, November 1999.

[mpi] MPI research reports. data.mpi-sb.mpg.de/internet/reports.nsf.

[MS96] David R. Musser and Atul Saini. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley professional computing ser. Addison-Wesley, Reading, MA, 1996.

[M.T73] M.T. McClellan. The Exact Solution of Systems of Linear Equations with Polynomial Coefficients. *JACM*, 20(4):563–588, October 1973.

[Mye95] N. Myers. A new and usefull template technique: traits. *C++ Report*, 6:34–35, 1995.

[MZ94] M. Müller and J. Ziegler. An implementation of a convex hull algorithm. Technical Report MPI-I-94-105, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[Nef78] W. Nef. *Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergraphik*. Herbert Lang & Cie AG, Bern, 1978.

[NS90] W. Nef and P.-M. Schmidt. Computing a sweeping-plane in regular ("general") position: a numerical and a symbolic solution. *Journal of Symbolic Computation*, 10(6):633–646, december 1990.

[OTU87] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In Derick Wood, editor, *Proceedings of the 3rd Annual Symposium on Computational Geometry (SCG '87)*, pages 119–125, Waterloo, ON, Canada, June 1987. ACM Press.

[Ove96] M. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, volume 1148 of *LNCS*, pages 53–58. Springer, 1996.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry : An Introduction*. Springer, 1985.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual : the definitive reference to the UML from the original designers*. The Addison-Wesley object technology series. Addison-Wesley, 1999.

[RO90] J.R. Rossignac and M.A. O'Connor. SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 145–180. Elsevier Science Publishers B.V., North Holland, 1990.

[RSV84] H.-J. Reiffen, G. Scheja, and U. Vetter. *Algebra*, volume 110. BI Hochschultaschenbücher, Mannheim Wien Zürich, 1984.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, New York, Brisbane, Toronto, Singapore, 1986.

[Sch99] S. Schirra. Precision and robustness issues in geometric computation. In *Handbook on Computational Geometry*. Elsevier, 1999.

[Sch01] Jörg Schwerdt. *Entwurf von Optimierungsalgorithmen für geometrische Probleme im Bereich Rapid Prototyping und Manufacturing*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany, 2001.

[See96] M. Seel. A runtime test of integer arithmetic and linear algebra in LEDA. Technical Report MPI-I-96-1-033, Max-Planck-Institut für Informatik, Saarbrücken, 1996.

[See01] M. Seel. Implementation of planar nef polyhedra. Technical Report MPI-I-2001-1-003, Max-Planck-Institut für Informatik, Saarbrücken, 2001.

[Sto91] J. Stolfi. *Oriented projective geometry : a framework for geometric computations*. Academic Press, 1991.

[SVY00] S. Schirra, R. Veltkamp, and M. Yvinec. *The CGAL User Manual*, January 2000.

[Tar83] R.E. Tarjan. *Data structures and network algorithms*, volume 44. SIAM, 6th pr. 1991 edition, 1983.

[TV97] Roberto Tamassia and Luca Vismara. A case study in algorithm engineering for geometric computing. Technical Report CS-97-18, Department of Computer Science, Brown University, December 1997.

[Wol96] S. Wolfram. *The Mathematica Book, 3rd ed.* Wolfram Media/Cambridge University Press, 1996.

[Yap93] C.K. Yap. Towards exact geometric computation. In *Proceedings of the 5th Canadian Conference on Computational Geometry (CCCG'93)*, pages 405–419, 1993.

[Yap97] C.K. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, 1997.

[YD95] C.K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

# Appendix

## 4.1 Manual pages of the higher-dimensional Kernel

### 4.1.1 Linear Algebra on RT ( Linear_algebraHd )

**1. Definition**

The data type *Linear_algebraHd*<*RT*> encapsulates two classes *Matrix, Vector* and many functions of basic linear algebra. It is parametrized by a number type *RT*. An instance of data type *Matrix* is a matrix of variables of type *RT*, the so called ring type. Accordingly, *Vector* implements vectors of variables of type *RT*. The arithmetic type *RT* is required to behave like integers in the mathematical sense. The manual pages of *Vector* and *Matrix* follow below.

All functions compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector $c$ such that $c^T A = 0$ and $c^T b \neq 0$. All internal correctness checks can be switched on by the flag `CGAL_LA_SELFTEST`.

**2. Types**

*Linear_algebraHd*<*RT*>::*RT* — the ring type of the components.

*Linear_algebraHd*<*RT*>::*Vector* — the vector type.

*Linear_algebraHd*<*RT*>::*Matrix* — the matrix type.

*Linear_algebraHd*<*RT*>::*allocator_type* — the allocator used for memory management. *Linear_algebraHd*<*RT*> is an abbreviation for *Linear_algebraHd*<*RT*, *ALLOC* = *allocator*<*RT*,*LA*> >. Thus *allocator_type* defaults to the standard allocator offered by the STL.

**3. Operations**

*Matrix* *Linear_algebraHd*<*RT*>::transpose(*Matrix M*)

returns $M^T$ ($m \times n$ - matrix).

*bool* *Linear_algebraHd*<*RT*>::inverse(*Matrix M*, *Matrix*& *I*, *RT*& *D*, *Vector*& *c*)

determines whether $M$ has an inverse. It also computes either the inverse as $(1/D) \cdot I$ or when no inverse exists, a vector $c$ such that $c^T \cdot M = 0$.

*Matrix* *Linear_algebraHd<RT>*::inverse(*Matrix M*, *RT& D*)

returns the inverse matrix of $M$. More precisely, $1/D$ times the matrix returned is the inverse of $M$.
*Precondition*: *determinant*($M$) != 0.

*RT* *Linear_algebraHd<RT>*::determinant(*Matrix M*, *Matrix& L*, *Matrix& U*, *std::vector<int>& q*, *Vector& c*)

returns the determinant $D$ of $M$ and suffi cient information to verify that the value of the determinant is correct. If the determinant is zero then $c$ is a vector such that $c^T \cdot M = 0$. If the determinant is non-zero then $L$ and $U$ are lower and upper diagonal matrices respectively and $q$ encodes a permutation matrix $Q$ with $Q(i,j) = 1$ iff $i = q(j)$ such that $L \cdot M \cdot Q = U$, $L(0,0) = 1$, $L(i,i) = U(i-1,i-1)$ for all $i$, $1 \leq i < n$, and $D = s \cdot U(n-1,n-1)$ where $s$ is the determinant of $Q$. *Precondition*: $M$ is square.

*bool* *Linear_algebraHd<RT>*::verify_determinant(*Matrix M*, *RT D*, *Matrix& L*, *Matrix& U*, *std::vector<int> q*, *Vector& c*)

verifi es the conditions stated above.

*RT* *Linear_algebraHd<RT>*::determinant(*Matrix M*)

returns the determinant of $M$. *Precondition*: $M$ is square.

*int* *Linear_algebraHd<RT>*::sign_of_determinant(*Matrix M*)

returns the sign of the determinant of $M$. *Precondition*: $M$ is square.

*bool* *Linear_algebraHd<RT>*::linear_solver(*Matrix M*, *Vector b*, *Vector& x*, *RT& D*, *Matrix& spanning_vectors*, *Vector& c*)

determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $(1/D)x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system. *Precondition*: *M.row_dimension*( ) = *b.dimension*( ).

*bool* *Linear_algebraHd<RT>*::linear_solver(*Matrix M*, *Vector b*, *Vector& x*, *RT& D*, *Vector& c*)

determines whether the linear system $M \cdot x = b$ is solvable. If yes, then $(1/D)x$ is a solution, if not then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. *Precondition*: *M.row_dimension*( ) = *b.dimension*( ).

*bool* *Linear_algebraHd<RT>*::linear_solver(*Matrix M*, *Vector b*, *Vector& x*, *RT& D*)

as above, but without the witness *c Precondition*: *M.row_dimension*( ) = *b.dimension*( ).

*bool* *Linear_algebraHd<RT>*::is_solvable(*Matrix M*, *Vector b*)

determines whether the system $M \cdot x = b$ is solvable
*Precondition*: *M.row_dimension*( ) = *b.dimension*( ).

*bool* *Linear_algebraHd<RT>*::homogeneous_linear_solver(*Matrix M*, *Vector& x*)

determines whether the homogeneous linear system $M \cdot x = 0$ has a non - trivial solution. If yes, then $x$ is such a solution.

*int* *Linear_algebraHd<RT>*::homogeneous_linear_solver(*Matrix M*, *Matrix& spanning_vecs*)

determines the solution space of the homogeneous linear system $M \cdot x = 0$. It returns the dimension of the solution space. Moreover the columns of *spanning_vecs* span the solution space.

*int* *Linear_algebraHd<RT>*::independent_columns(*Matrix M*, *std::vector<int>& columns*)

returns the indices of a maximal subset of independent columns of $M$.

*int* *Linear_algebraHd<RT>*::rank(*Matrix M*)

returns the rank of matrix $M$

**4. Implementation**

The datatype *Linear_algebraHd<RT>* is a wrapper class for the linear algebra functionality on matrices and vectors. Operations *determinant*, *inverse*, *linear_solver*, and *rank* take time $O(n^3)$, and all other operations take time $O(nm)$. These time bounds ignore the cost for multiprecision arithmetic operations.

All functions on integer matrices compute the exact result, i.e., there is no rounding error. The implemenation follows a proposal of J. Edmonds (J. Edmonds, Systems of distinct representatives and linear algebra, Journal of Research of the Bureau of National Standards, (B), 71, 241 - 245). Most functions of linear algebra are *checkable* , i.e., the programs can be asked for a proof that their output is correct. For example, if the linear system solver declares a linear system $Ax = b$ unsolvable it also returns a vector $c$ such that $c^T A = 0$ and $c^T b \neq 0$.

## Vectors with NT Entries ( Vector )

**1. Definition**

An instance of data type *Vector* is a vector of variables of number type *NT*. Together with the type *Matrix* it realizes the basic operations of linear algebra.

**2. Types**

| | |
|---|---|
| *Vector*::*NT* | the ring type of the components. |
| *Vector*::*iterator* | the iterator type for accessing components. |
| *Vector*::*const_iterator* | the const iterator type for accessing components. |

**3. Creation**

*Vector v*; creates an instance *v* of type *Vector*.

*Vector v*(*int d*); creates an instance *v* of type *Vector*. *v* is initialized to a vector of dimension *d*.

*Vector v*(*int d*, *NT x*);
creates an instance *v* of type *Vector*. *v* is initialized to a vector of dimension *d* with entries *x*.

template <*class Forward_iterator*>
*Vector v*(*Forward_iterator first*, *Forward_iterator last*);
creates an instance *v* of type *Vector*; *v* is initialized to the vector with entries *set* $[first, last)$. *Requirement*: *Forward_iterator* has value type *NT*.

**4. Operations**

| | | |
|---|---|---|
| *int* | *v*.dimension() | returns the dimension of *v*. |
| *bool* | *v*.is_zero() | returns true iff *v* is the zero vector. |
| *NT*& | *v*[*int i*] | returns *i*-th component of *v*. *Precondition*: $0 \leq i \leq v.dimension() - 1$. |
| *iterator* | *v*.begin() | iterator to the first component. |
| *iterator* | *v*.end() | iterator beyond the last component. |

The same operations *begin*( ), *end*( ) exist for *const_iterator*.

| | | |
|---|---|---|
| *Vector* | $v + v1$ | Addition. *Precondition*: $v.dimension() == v1.dimension()$. |
| *Vector* | $v - v1$ | Subtraction. *Precondition*: $v.dimension() = v1.dimension()$. |

| | | |
|---|---|---|
| *NT* | $v * v1$ | Inner Product. *Precondition*: $v.dimension() = v1.dimension()$. |
| *Vector* | $-v$ | Negation. |
| *Vector&* | $v += v1$ | Addition plus assignment. *Precondition*: $v.dimension() == v1.dimension()$. |
| *Vector&* | $v -= v1$ | Subtraction plus assignment. *Precondition*: $v.dimension() == v1.dimension()$. |
| *Vector&* | $v *= NT\ s$ | Scalar multiplication plus assignment. |
| *Vector&* | $v /= NT\ s$ | Scalar division plus assignment. |
| *Vector* | $NT\ r * v$ | Componentwise multiplication with number *r*. |
| *Vector* | $v * NT\ r$ | Componentwise multiplication with number *r*. |

## Matrices with NT Entries ( Matrix )

### 1. Definition

An instance of data type *Matrix* is a matrix of variables of number type *NT*. The types *Matrix* and *Vector* together realize many functions of basic linear algebra.

### 2. Types

| | |
|---|---|
| *Matrix*::*NT* | the ring type of the components. |
| *Matrix*::*iterator* | bidirectional iterator for accessing components. |
| *Matrix*::*row_iterator* | random access iterator for accessing row entries. |
| *Matrix*::*column_iterator* | random access iterator for accessing column entries. |

There are also constant versions of the above iterators: *const_iterator*, *row_const_iterator*, and *column_const_iterator*.

| | |
|---|---|
| *Matrix*::*Identity* | a tag class for identity initialization |
| *Matrix*::*Vector* | the vector type used. |

### 3. Creation

| | |
|---|---|
| *Matrix M*; | creates an instance *M* of type *Matrix*. |
| *Matrix M*(*int n*); | creates an instance *M* of type *Matrix* of dimension $n \times n$ initialized to the zero matrix. |
| *Matrix M*(*int m*, *int n*); | creates an instance *M* of type *Matrix* of dimension $m \times n$ initialized to the zero matrix. |
| *Matrix M*(*std*::*pair*<*int*,*int*> *p*); | creates an instance *M* of type *Matrix* of dimension *p.first*$\times$*p.second* initialized to the zero matrix. |
| *Matrix M*(*int n*, *Identity*, $NT\ x = NT(1)$); | creates an instance *M* of type *Matrix* of dimension $n \times n$ initialized to the identity matrix (times *x*). |
| *Matrix M*(*int m*, *int n*, *NT x*); | creates an instance *M* of type *Matrix* of dimension $m \times n$ initialized to the matrix with *x* entries. |

template <*class Forward_iterator*>
*Matrix M*(*Forward_iterator first*, *Forward_iterator last*);

creates an instance $M$ of type *Matrix*. Let $S$ be the ordered set of $n$ column-vectors of common dimension $m$ as given by the iterator range $[first, last)$. $M$ is initialized to an $m \times n$ matrix with the columns as specified by $S$. *Precondition*: *Forward_iterator* has a value type $V$ from which we require to provide a iterator type $V$::*const_iterator*, to have $V$::*value_type* $== NT$. Note that *Vector* or *std*::*vector*<*NT*> fulfill these requirements.

*Matrix M*(*std*::*vector*< *Vector* > *A*);

creates an instance $M$ of type *Matrix*. Let $A$ be an array of $n$ column-vectors of common dimension $m$. $M$ is initialized to an $m \times n$ matrix with the columns as specified by $A$.

**4. Operations**

| | | |
|---|---|---|
| *int* | *M*.row_dimension( ) | returns $n$, the number of rows of $M$. |
| *int* | *M*.column_dimension( ) | returns $m$, the number of columns of $M$. |
| *std*::*pair*<*int*, *int*> | *M*.dimension( ) | returns $(m,n)$, the dimension pair of $M$. |
| *Vector*& | *M*.row(*int i*) | returns the $i$-th row of $M$ (an $m$ - vector). *Precondition*: $0 \le i \le m-1$. |
| *Vector* | *M*.column(*int i*) | returns the $i$-th column of $M$ (an $n$ - vector). *Precondition*: $0 \le i \le n-1$. |
| *NT*& | *M*(*int i*, *int j*) | returns $M_{i,j}$. *Precondition*: $0 \le i \le m-1$ and $0 \le j \le n-1$. |
| *void* | *M*.swap_rows(*int i*, *int j*) | swaps rows $i$ and $j$. *Precondition*: $0 \le i \le m-1$ and $0 \le j \le m-1$. |
| *void* | *M*.swap_columns(*int i*, *int j*) | swaps columns $i$ and $j$. *Precondition*: $0 \le i \le n-1$ and $0 \le j \le n-1$. |
| *row_iterator* | *M*.row_begin(*int i*) | an iterator pointing to the first entry of the $i$th row. *Precondition*: $0 \le i \le m-1$. |
| *row_iterator* | *M*.row_end(*int i*) | an iterator pointing beyond the last entry of the $i$th row. *Precondition*: $0 \le i \le m-1$. |
| *column_iterator* | *M*.column_begin(*int i*) | an iterator pointing to the first entry of the $i$th column. *Precondition*: $0 \le i \le n-1$. |
| *column_iterator* | *M*.column_end(*int i*) | an iterator pointing beyond the last entry of the $i$th column. *Precondition*: $0 \le i \le n-1$. |

The same operations exist for *row_const_iterator* and *column_const_iterator*.

| | | |
|---|---|---|
| *bool* | *M* == *M1* | Test for equality. |
| *bool* | *M* != *M1* | Test for inequality. |

**Arithmetic Operators**

| | | |
|---|---|---|
| *Matrix* | *M* + *M1* | Addition. *Precondition*: *M.row_dimension*( ) == *M1.row_dimension*( ) and *M.column_dimension*( ) == *M1.column_dimension*( ) |
| *Matrix* | *M* − *M1* | Subtraction. *Precondition*: *M.row_dimension*( ) == *M1.row_dimension*( ) and *M.column_dimension*( ) == *M1.column_dimension*( ) |

| | | |
|---|---|---|
| *Matrix* | $-M$ | Negation. |
| *Matrix* | $M * M1$ | Multiplication. *Precondition*: *M.column_dimension*( ) = *M1.row_dimension*( ). |
| *Vector* | $M *$ *Vector vec* | Multiplication with vector. *Precondition*: *M.column_dimension*( ) = *vec.dimension*( ). |
| *Matrix* | $NT\ x * M$ | Multiplication of every entry with *x*. |
| *Matrix* | $M * NT\ x$ | Multiplication of every entry with *x*. |

### 4.1.2 Points in d-space ( Point_d )

**1. Definition**

An instance of data type *Point_d<R>* is a point of Euclidean space in dimension $d$. A point $p = (p_0, \ldots, p_{d-1})$ in $d$-dimensional space can be represented by homogeneous coordinates $(h_0, h_1, \ldots, h_d)$ of number type $RT$ such that $p_i = h_i/h_d$, which is of type $FT$. The homogenizing coordinate $h_d$ is positive.

We call $p_i$, $0 \leq i < d$ the $i$-th Cartesian coordinate and $h_i$, $0 \leq i \leq d$, the $i$-th homogeneous coordinate. We call $d$ the dimension of the point.

**2. Types**

*Point_d<R>::RT* the ring type.

*Point_d<R>::FT* the field type.

*Point_d<R>::LA* the linear algebra layer.

*Point_d<R>::Cartesian_const_iterator*
a read-only iterator for the cartesian coordinates.

*Point_d<R>::Homogeneous_const_iterator*
a read-only iterator for the homogeneous coordinates.

**3. Creation**

*Point_d<R> p(int d = 0);* introduces a variable $p$ of type *Point_d<R>* in $d$-dimensional space.

*Point_d<R> p(int d, Origin);*
introduces a variable $p$ of type *Point_d<R>* in $d$-dimensional space, initialized to the origin.

template <*class InputIterator*>
*Point_d<R> p(int d, InputIterator first, InputIterator last);*
introduces a variable $p$ of type *Point_d<R>* in dimension $d$. If *size* $[first, last) == d$ this creates a point with Cartesian coordinates *set* $[first, last)$. If *size* $[first, last) == p+1$ the range specifies the homogeneous coordinates $H$ = *set* $[first, last) = (\pm h_0, \pm h_1, \ldots, \pm h_d)$ where the sign chosen is the sign of $h_d$. *Precondition*: $d$ is nonnegative, $[first, last)$ has $d$ or $d+1$ elements where the last has to be non-zero, and the value type of *InputIterator* is $RT$.

template <*class InputIterator*>
*Point_d<R> p(int d, InputIterator first, InputIterator last, RT D);*
introduces a variable $p$ of type *Point_d<R>* in dimension $d$ initialized to the point with homogeneous coordinates as defined by $H$ = *set* $[first, last)$ and $D$: $(\pm H[0], \pm H[1], \ldots, \pm H[d-1], \pm D)$. The sign chosen is the sign of $D$. *Precondition*: $D$ is non-zero, the iterator range defines a $d$-tuple of $RT$, and the value type of *InputIterator* is $RT$.

*Point_d<R> p(RT x, RT y, RT w = 1);*
introduces a variable $p$ of type *Point_d<R>* in 2-dimensional space.

*Point_d<R> p(RT x, RT y, RT z, RT w);*
introduces a variable $p$ of type *Point_d<R>* in 3-dimensional space.

**4. Operations**

*int* $p$.dimension() returns the dimension of $p$.

| | | |
|---|---|---|
| *FT* | *p*.cartesian(*int i*) | returns the *i*-th Cartesian coordinate of *p*. *Precondition*: $0 \leq i < d$. |
| *FT* | *p*[*int i*] | returns the *i*-th Cartesian coordinate of *p*. *Precondition*: $0 \leq i < d$. |
| *RT* | *p*.homogeneous(*int i*) | returns the *i*-th homogeneous coordinate of *p*. *Precondition*: $0 \leq i \leq d$. |
| *Cartesian_const_iterator* | *p*.cartesian_begin() | returns an iterator pointing to the zeroth Cartesian coordinate $p_0$ of *p*. |
| *Cartesian_const_iterator* | *p*.cartesian_end() | returns an iterator pointing beyond the last Cartesian coordinate of *p*. |
| *Homogeneous_const_iterator* | *p*.homogeneous_begin() | returns an iterator pointing to the zeroth homogeneous coordinate $h_0$ of *p*. |
| *Homogeneous_const_iterator* | *p*.homogeneous_end() | returns an iterator pointing beyond the last homogeneous coordinate of *p*. |
| *Point_d*<*R*> | *p*.transform(*Aff_transformation_d*<*R*> *t*) | returns $t(p)$. |

**Arithmetic Operators, Tests and IO**

| | | |
|---|---|---|
| *Vector_d*<*R*> | $p - Origin\ o$ | returns the vector $\mathbf{Op}$. |
| *Vector_d*<*R*> | $p - q$ | returns $p - q$. *Precondition*: *p.dimension*( ) == *q.dimension*( ). |
| *Point_d*<*R*> | $p +$ *Vector_d*<*R*> *v* | returns $p + v$. *Precondition*: *p.dimension*( ) == *v.dimension*( ). |
| *Point_d*<*R*> | $p -$ *Vector_d*<*R*> *v* | returns $p - v$. *Precondition*: *p.dimension*( ) == *v.dimension*( ). |
| *Point_d*<*R*>& | $p +=$ *Vector_d*<*R*> *v* | adds *v* to *p*. *Precondition*: *p.dimension*( ) == *v.dimension*( ). |
| *Point_d*<*R*>& | $p -=$ *Vector_d*<*R*> *v* | subtracts *v* from *p*. *Precondition*: *p.dimension*( ) == *v.dimension*( ). |
| *bool* | $p ==$ *Origin* | returns true if *p* is the origin. |

**Downward compatibility**

We provide operations of the lower dimensional interface $x(\,), y(\,), z(\,), hx(\,), hy(\,), hz(\,), hw(\,)$.

**5. Implementation**

Points are implemented by arrays of *RT* items. All operations like creation, initialization, tests, point - vector arithmetic, input and output on a point *p* take time $O($*p.dimension*( )$)$. *dimension*( ), coordinate access and conversions take constant time. The space requirement for points is $O($*p.dimension*( )$)$.

### 4.1.3 Lines in d-space ( Line_d )

**1. Definition**

An instance of data type *Line_d* is an oriented line in $d$-dimensional Euclidian space.

**2. Types**

| | |
|---|---|
| *Line_d<RT>::R* | the representation type. |
| *Line_d<RT>::RT* | the ring type. |
| *Line_d<RT>::FT* | the field type. |
| *Line_d<RT>::LA* | the linear algebra layer. |

**3. Creation**

*Line_d<RT> l*(*int d* = 0);

introduces a variable $l$ of type *Line_d<RT>* and initializes it to some line in $d$ - dimensional space

*Line_d<RT> l*(*Point_d<R> p*, *Point_d<R> q*);

introduces a line through $p$ and $q$ and oriented from $p$ to $q$. *Precondition*: $p$ and $q$ are distinct and have the same dimension.

*Line_d<RT> l*(*Point_d<R> p*, *Direction_d<R> dir*);

introduces a line through $p$ with direction *dir*. *Precondition*: $p$ and *dir* have the same dimension, *dir* is not trivial.

*Line_d<RT> l*(*Segment_d<R> s*);

introduces a variable $l$ of type *Line_d<RT>* and initializes it to the line through *s.source*( ) and *s.target*( ) with direction from *s.source*( ) to *s.target*( ). *Precondition*: $s$ is not degenerate.

*Line_d<RT> l*(*Ray_d<R> r*);

introduces a variable $l$ of type *Line_d<RT>* and initializes it to the line through $r.point(1)$ and $r.point(2)$.

**4. Operations**

| | | |
|---|---|---|
| *int* | $l$.dimension( ) | returns the dimension of the underlying space. |
| *Point_d<R>* | $l$.point(*int i*) | returns an arbitrary point on $l$. It holds that $point(i) == point(j)$, iff $i == j$. Furthermore, $l$ is directed from $point(i)$ to $point(j)$, for all $i < j$. |
| *Line_d<R>* | $l$.opposite( ) | returns the line $(point(2), point(1))$. |
| *Direction_d<R>* | $l$.direction( ) | returns the direction of $l$. |
| *Line_d<R>* | $l$.transform(*Aff_transformation_d<R> t*) | returns $t(l)$. |
| *Line_d<R>* | $l + Vector\_d<R>\ v$ | returns $l + v$, i.e., $l$ translated by vector $v$. |
| *Point_d<R>* | $l$.projection(*Point_d<R> p*) | returns the point of intersection of $l$ with the hyperplane that is orthogonal to $l$ through $p$. |
| *bool* | $l$.has_on(*Point_d<R> p*) | returns true if $p$ lies on $l$ and false otherwise. |

**Non-Member Functions**

template <*class R*>
*bool* weak_equality(*Line_d*<*R*> *l1*, *Line_d*<*R*> *l2*)

Test for equality as unoriented lines.

template <*class R*>
*bool* parallel(*Line_d*<*R*> *l1*, *Line_d*<*R*> *l2*)

returns true if *l1* and *l2* are parallel as unoriented lines and false otherwise.

**5. Implementation**

Lines are implemented by a pair of points as an item type. All operations like creation, initialization, tests, direction calculation, input and output on a line $l$ take time $O(l.dimension(\,))$. *dimension*( ), coordinate and point access, and identity test take constant time. The operations for intersection calculation also take time $O(l.dimension(\,))$. The space requirement is $O(l.dimension(\,))$.

### 4.1.4 Affine Transformations ( Aff_transformation_d )

**1. Definition**

An instance of the data type *Aff_transformation_d*<*R*> is an affine transformation of $d$-dimensional space. It is specified by a square matrix $M$ of dimension $d+1$. All entries in the last row of $M$ except the diagonal entry must be zero; the diagonal entry must be non-zero. A point $p$ with homogeneous coordinates $(p[0],\ldots,p[d])$ can be transformed into the point $p.transform(A)$, where $A$ is an affine transformation created from $M$ by the constructors below.

**2. Types**

*Aff_transformation_d*<*R*>::*RT*

the ring type.

*Aff_transformation_d*<*R*>::*FT*

the field type.

*Aff_transformation_d*<*R*>::*LA*

the linear algebra layer.

**3. Creation**

*Aff_transformation_d*<*R*> *t*(*int d* = 0, *bool identity* = *false*);

introduces a transformation in $d$-dimensional space. If identity is true then the transformation is the identity transformation.

*Aff_transformation_d*<*R*> *t*(*typename LA*::*Matrix M*);

introduces the transformation of $d$ - space specified by matrix $M$. *Precondition*: $M$ is a square matrix of dimension $d+1$.

template <*typename Forward_iterator*>
*Aff_transformation_d*<*R*> *t*(*Forward_iterator start*, *Forward_iterator end*);

introduces the transformation of $d$-space specified by a diagonal matrix with entries *set* $[start,end)$ on the diagonal (a scaling of the space). *Precondition*: *set* $[start,end)$ is a vector of dimension $d+1$.

*Aff_transformation_d*<*R*> *t*(*Vector_d*<*R*> *v*);

introduces the translation by vector $v$.

*Aff_transformation_d*<*R*> *t*(*int d*, *RT num*, *RT den*);

returns a scaling by a scale factor $num/den$.

*Aff_transformation_d*<*R*> *t*(*int d*, *RT sin_num*, *RT cos_num*, *RT den*, *int e1* $= 0$, *int e2* $= 1$);

returns a planar rotation with sine and cosine values $sin\_num/den$ and $cos\_num/den$ in the plane spanned by the base vectors $b_{e1}$ and $b_{e2}$ in $d$-space. Thus the default use delivers a planar rotation in the *x-y* plane. *Precondition*: $sin\_num^2 + cos\_num^2 = den^2$ and $0 \leq e_1 < e_2 < d$

*Aff_transformation_d*<*R*> *t*(*int d*, *Direction_d*<*R*> *dir*, *RT num*, *RT den*, *int e1* $= 0$, *int e2* $= 1$);

returns a planar rotation within the plane spanned by the base vectors $b_{e1}$ and $b_{e2}$ in $d$-space. The rotation parameters are given by the 2-dimensional direction *dir*, such that the difference between the sines and cosines of the rotation given by *dir* and the approximated rotation are at most $num/den$ each.
*Precondition*: $dir.dimension() == 2$, $!dir.is\_degenerate()$ and $num < den$ is positive and $0 \leq e_1 < e_2 < d$

**4. Operations**

| | | |
|---|---|---|
| *int* | *t*.dimension() | the dimension of the underlying space |
| *typename LA*::*Matrix* | *t*.matrix() | returns the transformation matrix |
| *Aff_transformation_d*<*R*> | *t*.inverse() | returns the inverse transformation. *Precondition*: $t.matrix()$ is invertible. |
| *Aff_transformation_d*<*R*> | $t * s$ | composition of transformations. Note that transformations are not necessarily commutative. $t * s$ is the transformation which transforms first by $t$ and then by $s$. |

**5. Implementation**

Affine Transformations are implemented by matrices of integers as an item type. All operations like creation, initialization, input and output on a transformation $t$ take time $O(t.dimension()^2)$. $dimension()$ takes constant time. The operations for inversion and composition have the cubic costs of the used matrix operations. The space requirement is $O(t.dimension()^2)$.

**Linear and affine predicates**

For a iterator range $[first, last)$ we define $S = set\ [first, last)$ as the ordered tuple $(S[0], S[1], \ldots S[d-1])$ where $S[i] = *{+}{+}^{(i)} first$ (the element obtained by $i$ times forwarding the iterator by operator $++$ and then dereferencing it to get the value to which it points). We write $d = size\ [first, last)$. This extends the syntax of random access iterators to input iterators. If we index the tuple as above then we require that ${+}{+}^{(d+1)} first == last$.

In the following we require the Iterators to be globally of value type *Point_d<R>*. Also if we are handed over an iterator range $[first, last)$, then all points in $S = set\ [first, last)$ have the same dimension *dim*.

template <*class ForwardIterator*>
*Orientation* orientation(*ForwardIterator first*, *ForwardIterator last*)

determines the orientation of the points in the set $A = set\ [first, last)$ where $A$ consists of $d+1$ points in $d$ - space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \ldots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the $i$-th point in $A$. *Precondition*: $size\ [first, last) == d+1$ and $A[i].dimension(\,) == d\ \forall 0 \le i \le d$. *Precondition*: value type of *ForwardIterator* is *Point_d<R>*.

template <*class R*, *class ForwardIterator*>
*Oriented_side* side_of_oriented_sphere(*ForwardIterator first*, *ForwardIterator last*, *Point_d<R> x*)

determines to which side of the sphere $S$ defined by the points in $A = set\ [first, last)$ the point $x$ belongs, where $A$ consists of $d+1$ points in $d$ - space. The positive side is determined by the positive sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 & 1 \\ lift(A[0]) & lift(A[1]) & \ldots & lift(A[d]) & lift(x) \end{vmatrix}$$

where for a point $p$ with cartesian coordinates $p_i$ we use $lift(p)$ to denote the $d+1$-dimensional point with cartesian coordinate vector $(p_0, \ldots, p_{d-1}, \sum_{0 \le i < d} p_i^2)$. If the points in $A$ are positively oriented then the positive side is the inside of the sphere and the negative side is the outside of the sphere. *Precondition*: value type of *ForwardIterator* is *Point_d<R>*.

template <*class R*, *class ForwardIterator*>
*Bounded_side* side_of_bounded_sphere(*ForwardIterator first*, *ForwardIterator last*, *Point_d<R> p*)

determines whether the point $p$ lies *ON_BOUNDED_SIDE*, *ON_BOUNDARY*, or *ON_UNBOUNDED_SIDE* of the sphere defined by the points in $A = set\ [first, last)$ where $A$ consists of $d+1$ points in $d$-space. *Precondition*: value type of *ForwardIterator* is *Point_d<R>* and $orientation(first, last) \neq ZERO$.

template <*class R*, *class ForwardIterator*>
*bool* contained_in_simplex(*ForwardIterator first*, *ForwardIterator last*, *Point_d<R> p*)

determines whether $p$ is contained in the simplex spanned by the points in $A = set[first, last)$. $A$ may consists of up to $d+1$ points. *Precondition*: value type of *ForwardIterator* is *Point_d<R>* and the points in $A$ are affinely independent.

template <*class R*, *class ForwardIterator*>
*bool* contained_in_affine_hull(*ForwardIterator first*, *ForwardIterator last*, *Point_d*<*R*> *p*)

determines whether $p$ is contained in the affine hull of the points in $A = set[first, last)$. *Precondition*: value type of *ForwardIterator* is *Point_d*<*R*>.

template <*class ForwardIterator*>
*int* affine_rank(*ForwardIterator first*, *ForwardIterator last*)

computes the affine rank of the points in $A = set[first, last)$. *Precondition*: value type of *ForwardIterator* is *Point_d*<*R*>.

template <*class ForwardIterator*>
*bool* affinely_independent(*ForwardIterator first*, *ForwardIterator last*)

decides whether the points in $A = set[first, last)$ are affinely independent. *Precondition*: value type of *ForwardIterator* is *Point_d*<*R*>.

template <*class R*>
*Comparison_result* compare_lexicographically(*Point_d*<*R*> *p1*, *Point_d*<*R*> *p2*)

implements the lexicographic order on the cartesian coordinate tuple of points.

template <*class R*, *class ForwardIterator*>
*bool* contained_in_linear_hull(*ForwardIterator first*, *ForwardIterator last*,
*Vector_d*<*R*> *x*)

determines whether $x$ is contained in the linear hull of the vectors in $A = set[first, last)$. *Precondition*: value type of *ForwardIterator* is *Vector_d*<*R*>.

template <*class ForwardIterator*>
*int* linear_rank(*ForwardIterator first*, *ForwardIterator last*)

computes the linear rank of the vectors in $A = set[first, last)$. *Precondition*: value type of *ForwardIterator* is *Vector_d*<*R*>.

template <*class ForwardIterator*>
*bool* linearly_independent(*ForwardIterator first*, *ForwardIterator last*)

decides whether the vectors in $A$ are linearly independent. *Precondition*: value type of *ForwardIterator* is *Vector_d*<*R*>.

template <*class ForwardIterator*, *class OutputIterator*>
*OutputIterator* linear_base(*ForwardIterator first*, *ForwardIterator last*, *OutputIterator result*)

computes a basis of the linear space spanned by the vectors in *set* $[first, last)$ and returns it via an iterator range starting in *result*. The returned iterator marks the end of the output. *Precondition*: value type of *ForwardIterator* is *Vector_d*<*R*>.

**Constructions**

template <*class R*>
*Point_d*<*R*> lift_to_paraboloid(*Point_d*<*R*> *p*)

returns $p = (x_0, \ldots, x_{d-1})$ lifted to the paraboloid of revolution .

template <*class R*>
*Point_d*<*R*> project_along_d_axis(*Point_d*<*R*> *p*)

returns $p$ projected along the $d$-axis onto the hyperspace spanned by the first $d-1$ standard base vectors.

template <*class R*>
*Point_d*<*R*> midpoint(*Point_d*<*R*> *p*, *Point_d*<*R*> *q*)

returns the midpoint of $p$ and $q$.

template <*class R*>
*typename R*::*FT* squared_distance(*Point_d*<*R*> *p*, *Point_d*<*R*> *q*)
returns the squared distance between *p* and *q*.

### 4.1.5 Convex Hulls ( Convex_hull_d )

**1. Definition**

An instance $C$ of type *Convex_hull_d<R>* is the convex hull of a multi-set $S$ of points in $d$-dimensional space. We call $S$ the underlying point set and $d$ or *dim* the dimension of the underlying space. We use *dcur* to denote the affine dimension of $S$. The data type supports incremental construction of hulls.

The closure of the hull is maintained as a simplicial complex, i.e., as a collection of simplices. The intersection of any two is a face of both[21]. In the sequel we reserve the word simplex for the simplices of dimension *dcur*. For each simplex there is a handle of type *Simplex_handlex* and for each vertex there is a handle of type *Vertex_handle*. Each simplex has $1 + dcur$ vertices indexed from 0 to *dcur*; for a simplex $s$ and an index $i$, $C.vertex(s,i)$ returns the $i$-th vertex of $s$. For any simplex $s$ and any index $i$ of $s$ there may or may not be a simplex $t$ opposite to the $i$-th vertex of $s$. The function $C.opposite\_simplex(s,i)$ returns $t$ if it exists and returns *Simplex_handle*( ) (the undefined handle) otherwise. If $t$ exists then $s$ and $t$ share *dcur* vertices, namely all but the vertex with index $i$ of $s$ and the vertex with index $C.index\_of\_vertex\_in\_opposite\_simplex(s,i)$ of $t$. Assume that $t$ exists and let $j = C.index\_of\_vertex\_in\_opposite\_simplex(s,i)$. Then $s = C.opposite\_simplex(t,j)$ and $i = C.index\_of\_vertex\_in\_opposite\_simplex(t,j)$.

The boundary of the hull is also a simplicial complex. All simplices in this complex have dimension $dcur - 1$. For each boundary simplex there is a handle of type *Facet_handle*. Each facet has *dcur* vertices indexed from 0 to $dcur - 1$. If $dcur > 1$ then for each facet $f$ and each index $i$, $0 \le i < dcur$, there is a facet $g$ opposite to the $i$-th vertex of $f$. The function $C.opposite\_facet(f,i)$ returns $g$. Two neighboring facets $f$ and $g$ share $dcur - 1$ vertices, namely all but the vertex with index $i$ of $f$ and the vertex with index $C.index\_of\_vertex\_in\_opposite\_facet(f,i)$ of $g$. Let $j = C.index\_of\_vertex\_in\_opposite\_facet(f,i)$. Then $f = C.opposite\_facet(g,j)$ and $i = C.index\_of\_vertex\_in\_opposite\_facet(g,j)$.

**2. Types**

| | |
|---|---|
| *Convex_hull_d<R>::R* | the representation class. |
| *Convex_hull_d<R>::Point_d* | the point type. |
| *Convex_hull_d<R>::Hyperplane_d* | the hyperplane type. |
| *Convex_hull_d<R>::Simplex_handle* | handle for simplices. |
| *Convex_hull_d<R>::Facet_handle* | handle for facets. |
| *Convex_hull_d<R>::Vertex_handle* | handle for vertices. |
| *Convex_hull_d<R>::Simplex_iterator* | iterator for simplices. |
| *Convex_hull_d<R>::Vertex_iterator* | iterator for vertices. |
| *Convex_hull_d<R>::Facet_iterator* | iterator for facets. |
| *Convex_hull_d<R>::Hull_vertex_iterator* | iterator for vertices that are part of the convex hull. |

Note that each iterator fits the handle concept, i.e. iterators can be used as handles. Note also that all iterator and handle types come also in a const flavor, e.g., *Vertex_const_iterator* is the constant version of *Vertex_iterator*. Thus use the const version whenever the the convex hull object is referenced as constant.

| | |
|---|---|
| *Convex_hull_d<R>::Point_const_iterator* | const iterator for all inserted points. |
| *Convex_hull_d<R>::Hull_point_const_iterator* | const iterator for all points of the hull. |

[21] The empty set if a facet of every simplex.

**3. Creation**

*Convex_hull_d*<*R*> *C*(*int d*, *R Kernel* = *R*( ));

creates an instance *C* of type *Convex_hull_d*. The dimension of the underlying space is *d* and *S* is initialized to the empty point set. The traits class *R* specifies the models of all types and the implementations of all geometric primitives used by the convex hull class. The default model is one of the *d*-dimensional representation classes (e.g., *Homogeneous_d*).

The data type *Convex_hull_d* offers neither copy constructor nor assignment operator.

**4. Operations**

All operations below that take a point *x* as argument have the common precondition that *x* is a point of ambient space.

| | | |
|---|---|---|
| *int* | *C*.dimension( ) | returns the dimension of ambient space |
| *int* | *C*.current_dimension( ) | returns the affine dimension *dcur* of *S*. |
| *Point_d* | *C*.associated_point(*Vertex_handle v*) | returns the point associated with vertex *v*. |
| *Vertex_handle* | *C*.vertex_of_simplex(*Simplex_handle s*, *int i*) | returns the vertex corresponding to the *i*-th vertex of *s*. *Precondition*: $0 \leq i \leq dcur$. |
| *Point_d* | *C*.point_of_simplex(*Simplex_handle s*, *int i*) | same as *C.associated_point*(*C.vertex_of_simplex*$(s,i)$). |
| *Simplex_handle* | *C*.opposite_simplex(*Simplex_handle s*, *int i*) | returns the simplex opposite to the *i*-th vertex of *s* (*Simplex_handle*( ) if there is no such simplex). *Precondition*: $0 \leq i \leq dcur$. |
| *int* | *C*.index_of_vertex_in_opposite_simplex(*Simplex_handle s*, *int i*) | returns the index of the vertex opposite to the *i*-th vertex of *s*. *Precondition*: $0 \leq i \leq dcur$ and there is a simplex opposite to the *i*-th vertex of *s*. |
| *Simplex_handle* | *C*.simplex(*Vertex_handle v*) | returns a simplex of which *v* is a node. Note that this simplex is not unique. |
| *int* | *C*.index(*Vertex_handle v*) | returns the index of *v* in *simplex*$(v)$. |
| *Vertex_handle* | *C*.vertex_of_facet(*Facet_handle f*, *int i*) | returns the vertex corresponding to the *i*-th vertex of $f$. *Precondition*: $0 \leq i < dcur$. |
| *Point_d* | *C*.point_of_facet(*Facet_handle f*, *int i*) | same as *C.associated_point*(*C.vertex_of_facet*$(f,i)$). |
| *Facet_handle* | *C*.opposite_facet(*Facet_handle f*, *int i*) | returns the facet opposite to the *i*-th vertex of $f$ (*Facet_handle*( ) if there is no such facet). *Precondition*: $0 \leq i < dcur$ and $dcur > 1$. |
| *int* | *C*.index_of_vertex_in_opposite_facet(*Facet_handle f*, *int i*) | returns the index of the vertex opposite to the *i*-th vertex of $f$. *Precondition*: $0 \leq i < dcur$ and $dcur > 1$. |

| | | |
|---|---|---|
| *Hyperplane_d* | *C*.hyperplane_supporting(*Facet_handle f*) | |
| | | returns a hyperplane supporting facet $f$. The hyperplane is oriented such that the interior of *C* is on the negative side of it. *Precondition*: $f$ is a facet of *C* and $dcur > 1$. |
| *Vertex_handle* | *C*.insert(*Point_d x*) | adds point *x* to the underlying set of points. If *x* is equal to (the point associated with) a vertex of the current hull this vertex is returned and its associated point is changed to *x*. If *x* lies outside the current hull, a new vertex *v* with associated point *x* is added to the hull and returned. In all other cases, i.e., if *x* lies in the interior of the hull or on the boundary but not on a vertex, the current hull is not changed and *Vertex_handle*( ) is returned. If *CGAL_CHECK_EXPENSIVE* is defined then the validity check *is_valid*(*true*) is executed as a post condition. |
| template <*typename Forward_iterator*> | | |
| *void* | *C*.insert(*Forward_iterator first*, *Forward_iterator last*) | |
| | | adds $S = set[first, last)$ to the underlying set of points. If any point $S[i]$ is equal to (the point associated with) a vertex of the current hull its associated point is changed to $S[i]$. |
| *bool* | *C*.is_dimension_jump(*Point_d x*) | |
| | | returns true if *x* is not contained in the affine hull of *S*. |
| *std*::*list*<*Facet_handle*> | *C*.facets_visible_from(*Point_d x*) | |
| | | returns the list of all facets that are visible from *x*. *Precondition*: *x* is contained in the affine hull of *S*. |
| *Bounded_side* | *C*.bounded_side(*Point_d x*) | returns *ON_BOUNDED_SIDE* (*ON_BOUNDARY*,*ON_UNBOUNDED_SIDE*) if *x* is contained in the interior (lies on the boundary, is contained in the exterior) of *C*. *Precondition*: *x* is contained in the affine hull of *S*. |
| *void* | *C*.clear(*int d*) | reinitializes *C* to an empty hull in *d*-dimensional space. |
| *int* | *C*.number_of_vertices( ) | returns the number of vertices of *C*. |
| *int* | *C*.number_of_facets( ) | returns the number of facets of *C*. |
| *int* | *C*.number_of_simplices( ) | returns the number of bounded simplices of *C*. |
| *void* | *C*.print_statistics( ) | gives information about the size of the current hull and the number of visibility tests performed. |
| *bool* | *C*.is_valid(*bool throw_exceptions = false*) | |
| | | checks the validity of the data structure. If *throw_exceptions*== *thrue* then the program throws the following exceptions to inform about the problem.<br>`chull_has_center_on_wrong_side_of_hull_facet` the hyperplane supporting a facet has the wrong orientation.<br>`chull_has_local_non_convexity` a ridge is locally non convex.<br>`chull_has_double_coverage` the hull has a winding number larger than 1. |

**Lists and Iterators**

| | | |
|---|---|---|
| *Vertex_iterator* | *C*.vertices_begin( ) | returns an iterator to start iteration over all vertices of *C*. |
| *Vertex_iterator* | *C*.vertices_end( ) | the past the end iterator for vertices. |

| | | |
|---|---|---|
| *Simplex_iterator* | *C*.simplices_begin( ) | returns an iterator to start iteration over all simplices of *C*. |
| *Simplex_iterator* | *C*.simplices_end( ) | the past the end iterator for simplices. |
| *Facet_iterator* | *C*.facets_begin( ) | returns an iterator to start iteration over all facets of *C*. |
| *Facet_iterator* | *C*.facets_end( ) | the past the end iterator for facets. |
| *Hull_vertex_iterator* | *C*.hull_vertices_begin( ) | returns an iterator to start iteration over all hull vertex of *C*. Remember that the hull is a simplicial complex. |
| *Hull_vertex_iterator* | *C*.hull_vertices_end( ) | the past the end iterator for hull vertices. |
| *Point_const_iterator* | *C*.points_begin( ) | returns the start iterator for all points that have been inserted to construct *C*. |
| *Point_const_iterator* | *C*.points_end( ) | returns the past the end iterator for all points. |
| *Hull_point_const_iterator* | *C*.hull_points_begin( ) | returns an iterator to start iteration over all inserted points that are part of the convex hull *C*. Remember that the hull is a simplicial complex. |
| *Hull_point_const_iterator* | *C*.hull_points_end( ) | returns the past the end iterator for points of the hull. |
| template <*typename Visitor*> *void* | *C*.visit_all_facets(*Visitor V*) | each facet of *C* is visited by the visitor object *V*. *V* has to have a function call operator: *void operator*( )(*Facet_handle*) *const* |
| *std*::*list*<*Point_d*> | *C*.all_points( ) | returns a list of all points in *C*. |
| *std*::*list*<*Vertex_handle*> | *C*.all_vertices( ) | returns a list of all vertices of *C* (also interior ones). |
| *std*::*list*<*Simplex_handle*> | *C*.all_simplices( ) | returns a list of all simplices in *C*. |
| *std*::*list*<*Facet_handle*> | *C*.all_facets( ) | returns a list of all facets of *C*. |

**Iteration Statements**

**forall_ch_vertices**($v$,$C$) { "the vertices of $C$ are successively assigned to $v$" }

**forall_ch_simplices**($s$,$C$) { "the simplices of $C$ are successively assigned to $s$" }

**forall_ch_facets**($f$,$C$) { "the facets of $C$ are successively assigned to $f$" }

**5. Implementation**

The implementation of type *Convex_hull_d* is based on [CMS93] and [BMS94]. The details of the implementation can be found in the implementation document available at the download site of this package.

The time and space requirements are input dependent. Let $C_1$, $C_2$, $C_3$, ... be the sequence of hulls constructed and for a point $x$ let $k_i$ be the number of facets of $C_i$ that are visible from $x$ and that are not already facets of $C_{i-1}$. Then the time for inserting $x$ is $O(dim \sum_i k_i)$ and the number of new simplices constructed during the insertion of $x$ is the number of facets of the hull which were not already facets of the hull before the insertion.

The data type *Convex_hull_d* is derived from *Regular_complex_d*. The space requirement of regular complexes is essentially $12(dim+2)$ bytes times the number of simplices plus the space for the points. *Convex_hull_d* needs an additional $8+(4+x)dim$ bytes per simplex where $x$ is the space requirement of the underlying number type and an additional 12 bytes per point. The total is therefore $(16+x)dim+32$ bytes times the number of simplices plus $28+x \cdot dim$ bytes times the number of points.

### 4.1.6 Delaunay Triangulations ( Delaunay_d )

**1. Definition**

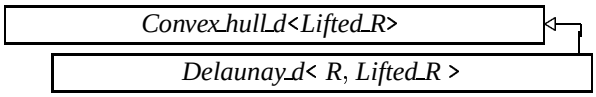
An instance *DT* of type *Delaunay_d< R, Lifted_R >* is the nearest and furthest site Delaunay triangulation of a set *S* of points in some *d*-dimensional space. We call *S* the underlying point set and *d* or *dim* the dimension of the underlying space. We use *dcur* to denote the affine dimension of *S*. The data type supports incremental construction of Delaunay triangulations and various kind of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

A Delaunay triangulation is a simplicial complex. All simplices in the Delaunay triangulation have dimension *dcur*. In the nearest site Delaunay triangulation the circumsphere of any simplex in the triangulation contains no point of *S* in its interior. In the furthest site Delaunay triangulation the circumsphere of any simplex contains no point of *S* in its exterior. If the points in *S* are co-circular then any triangulation of *S* is a nearest as well as a furthest site Delaunay triangulation of *S*. If the points in *S* are not co-circular then no simplex can be a simplex of both triangulations. Accordingly, we view *DT* as either one or two collection(s) of simplices. If the points in *S* are co-circular there is just one collection: the set of simplices of some triangulation. If the points in *S* are not co-circular there are two collections. One collection consists of the simplices of a nearest site Delaunay triangulation and the other collection consists of the simplices of a furthest site Delaunay triangulation.

For each simplex of maximal dimension there is a handle of type *Simplex_handle* and for each vertex of the triangulation there is a handle of type *Vertex_handle*. Each simplex has $1 + dcur$ vertices indexed from 0 to *dcur*. For any simplex *s* and any index *i*, $DT.vertex\_of(s,i)$ returns the *i*-th vertex of *s*. There may or may not be a simplex *t* opposite to the vertex of *s* with index *i*. The function $DT.opposite\_simplex(s,i)$ returns *t* if it exists and returns *Simplex_handle*( ) otherwise. If *t* exists then *s* and *t* share *dcur* vertices, namely all but the vertex with index *i* of *s* and the vertex with index $DT.index\_of\_vertex\_in\_opposite\_simplex(s,i)$ of *t*. Assume that $t = DT.opposite\_simplex(s,i)$ exists and let $j = DT.index\_of\_vertex\_in\_opposite\_simplex(s,i)$. Then $s = DT.opposite\_simplex(t,j)$ and $i = DT.index\_of\_vertex\_in\_opposite\_simplex(t,j)$. In general, a vertex belongs to many simplices.

Any simplex of *DT* belongs either to the nearest or to the furthest site Delaunay triangulation or both. The test $DT.simplex\_of\_nearest(dt\_simplex\ s)$ returns true if *s* belongs to the nearest site triangulation and the test $DT.simplex\_of\_furthest(dt\_simplex\ s)$ returns true if *s* belongs to the furthest site triangulation.

**2. Generalization**




**3. Types**

| | |
|---|---|
| *Delaunay_d< R, Lifted_R >::Simplex_handle* | handles to the simplices of the complex. |
| *Delaunay_d< R, Lifted_R >::Vertex_handle* | handles to vertices of the complex. |
| *Delaunay_d< R, Lifted_R >::Point_d* | the point type |
| *Delaunay_d< R, Lifted_R >::Sphere_d* | the sphere type |
| *Delaunay_d< R, Lifted_R >::Delaunay_voronoi_kind* { NEAREST, FURTHEST } | interface flags |

To use these types you can typedef them into the global scope after instantiation of the class. We use *Vertex_handle* instead of *Delaunay_d< R, Lifted_R >:: Vertex_handle* from now on. Similarly we use *Simplex_handle*.

| | |
|---|---|
| *Delaunay_d< R, Lifted_R >::Point_const_iterator* | the iterator for points. |
| *Delaunay_d< R, Lifted_R >::Vertex_iterator* | the iterator for vertices. |
| *Delaunay_d< R, Lifted_R >::Simplex_iterator* | the iterator for simplices. |

**4. Creation**

*Delaunay_d*< *R*, *Lifted_R* > *DT*(*int d*, *R k1* = *R*( ), *Lifted_R k2* = *Lifted_R*( ));

creates an instance *DT* of type *Delaunay_d*. The dimension of the underlying space is *d* and *S* is initialized to the empty point set. The traits class *R* specifies the models of all types and the implementations of all geometric primitives used by the Delaunay class. The traits class *Lifted_R* specifies the models of all types and the implementations of all geometric primitives used by the base class of *Delaunay_d*< *R*, *Lifted_R* >. The second template parameter defaults to the first: *Delaunay_d*<*R*> = *Delaunay_d*<*R*, *Lifted_R* = *R* >.

Both template arguments have to be models that fit a subset of requirements of the d-dimensional kernel. We list them at the end of this manual page.

The data type *Delaunay_d* offers neither copy constructor nor assignment operator.

**5. Operations**

All operations below that take a point *x* as an argument have the common precondition that *x.dimension*( ) = *DT.dimension*( ).

*int* *DT*.dimension() returns the dimension of ambient space

*int* *DT*.current_dimension()

returns the affine dimension of the current point set, i.e., $-1$ is *S* is empty, 0 if *S* consists of a single point, 1 if all points of *S* lie on a common line, etcetera.

*bool* *DT*.is_simplex_of_nearest(*Simplex_handle s*)

returns true if *s* is a simplex of the nearest site triangulation.

*bool* *DT*.is_simplex_of_furthest(*Simplex_handle s*)

returns true if *s* is a simplex of the furthest site triangulation.

*Vertex_handle* *DT*.vertex_of_simplex(*Simplex_handle s*, *int i*)

returns the vertex associated with the *i*-th node of *s*. *Precondition*: $0 \leq i \leq dcur$.

*Point_d* *DT*.associated_point(*Vertex_handle v*)

returns the point associated with vertex *v*.

*Point_d* *DT*.point_of_simplex(*Simplex_handle s*, *int i*)

returns the point associated with the *i*-th vertex of *s*. *Precondition*: $0 \leq i \leq dcur$.

*Simplex_handle* *DT*.opposite_simplex(*Simplex_handle s*, *int i*)

returns the simplex opposite to the *i*-th vertex of *s* (*Simplex_handle*( ) if there is no such simplex). *Precondition*: $0 \leq i \leq dcur$.

*int* *DT*.index_of_vertex_in_opposite_simplex(*Simplex_handle s*, *int i*)

returns the index of the vertex opposite to the *i*-th vertex of *s*. *Precondition*: $0 \leq i \leq dcur$.

*Simplex_handle* *DT*.simplex(*Vertex_handle v*)

returns a simplex of the nearest site triangulation incident to *v*.

*int* *DT*.index(*Vertex_handle v*)

returns the index of *v* in *DT.simplex*(*v*).

*bool* *DT*.contains(*Simplex_handle s*, *Point_d x*)

returns true if *x* is contained in the closure of simplex *s*.

| | | |
|---|---|---|
| *bool* | *DT*.empty( ) | decides whether *DT* is empty. |
| *void* | *DT*.clear( ) | reinitializes *DT* to the empty Delaunay triangulation. |
| *Vertex_handle* | *DT*.insert(*Point_d x*) | inserts point *x* into *DT* and returns the corresponding *Vertex_handle*. More precisely, if there is already a vertex $v$ in *DT* positioned at $x$ (i.e., $associated\_point(v)$ is equal to $x$) then $associated\_point(v)$ is changed to $x$ (i.e., $associated\_point(v)$ is made identical to $x$) and if there is no such vertex then a new vertex $v$ with $associated\_point(v) = x$ is added to *DT*. In either case, $v$ is returned. |
| *Simplex_handle* | *DT*.locate(*Point_d x*) | returns a simplex of the nearest site triangulation containing $x$ in its closure (returns *Simplex_handle*( ) if $x$ lies outside the convex hull of $S$). |
| *Vertex_handle* | *DT*.lookup(*Point_d x*) | if *DT* contains a vertex $v$ with $associated\_point(v) = x$ the result is $v$ otherwise the result is *Vertex_handle*( ). |
| *Vertex_handle* | *DT*.nearest_neighbor(*Point_d x*) | computes a vertex $v$ of *DT* that is closest to $x$, i.e., $dist(x, associated\_point(v)) = \min\{dist(x, associated\_point(u)) \mid u \in S\}$. |
| *std*::*list*<*Vertex_handle*> | *DT*.range_search(*Sphere_d C*) | returns the list of all vertices contained in the closure of sphere $C$. |
| *std*::*list*<*Vertex_handle*> | *DT*.range_search(*std*::*vector*<*Point_d*> *A*) | returns the list of all vertices contained in the closure of the simplex whose corners are given by $A$. *Precondition*: $A$ must consist of $d+1$ affinely independent points in base space. |
| *std*::*list*<*Simplex_handle*> | *DT*.all_simplices(*Delaunay_voronoi_kind k = NEAREST*) | returns a list of all simplices of either the nearest or the furthest site Delaunay triangulation of $S$. |
| *std*::*list*<*Vertex_handle*> | *DT*.all_vertices(*Delaunay_voronoi_kind k = NEAREST*) | returns a list of all vertices of either the nearest or the furthest site Delaunay triangulation of $S$. |
| *std*::*list*<*Point_d*> | *DT*.all_points( ) | returns $S$. |
| *Point_const_iterator* | *DT*.points_begin( ) | returns the start iterator for points in *DT*. |
| *Point_const_iterator* | *DT*.points_end( ) | returns the past the end iterator for points in *DT*. |
| *Simplex_iterator* | *DT*.simplices_begin(*Delaunay_voronoi_kind k = NEAREST*) | returns the start iterator for simplices of *DT*. |
| *Simplex_iterator* | *DT*.simplices_end( ) | returns the past the end iterator for simplices of *DT*. |

**6. Implementation**

The data type is derived from *Convex_hull_d* via the lifting map. For a point $x$ in $d$-dimensional space let $lift(x)$ be its lifting to the unit paraboloid of revolution. There is an intimate relationship between the Delaunay triangulation of a point set $S$ and the convex hull of $lift(S)$: The nearest site Delaunay triangulation is the

projection of the lower hull and the furthest site Delaunay triangulation is the upper hull. For implementation details we refer the reader to the implementation report available from the CGAL server.

The space requirement is the same as for convex hulls. The time requirement for an insert is the time to insert the lifted point into the convex hull of the lifted points.

**7. Example**

The abstract data type *Delaunay_d* has a default instantiation by means of the *d*-dimensional geometric kernel.

```
#include <CGAL/Homogeneous_d.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Delaunay_d.h>

typedef leda_integer RT;
typedef CGAL::Homogeneous_d<RT> Kernel;
typedef CGAL::Delaunay_d<Kernel> Delaunay_d;
typedef Delaunay_d::Point_d Point;
typedef Delaunay_d::Simplex_handle Simplex_handle;
typedef Delaunay_d::Vertex_handle Vertex_handle;

int main()
{
  Delaunay_d T(2);
  Vertex_handle v1 = T.insert(Point_d(2,11));
  ...
}
```

## 4.2 Manual pages of the Nef polyhedron package

### 4.2.1 Nef Polyhedra in the Plane ( Nef_polyhedron_2 )

**1. Definition**

An instance of data type *Nef_polyhedron_2<T>* is a subset of the plane that is the result of forming complements and intersections starting from a finite set $H$ of halfspaces. *Nef_polyhedron_2* is closed under all binary set operations *intersection, union, difference, complement* and under the topological operations *boundary, closure,* and *interior*.

The template parameter $T$ is specified via an extended kernel concept. $T$ must be a model of the concept *ExtendedKernelTraits_2*.

**2. Types**

| | |
|---|---|
| *Nef_polyhedron_2<T>::Line* | the oriented lines modeling halfplanes |
| *Nef_polyhedron_2<T>::Point* | the affine points of the plane. |
| *Nef_polyhedron_2<T>::Direction* | directions in our plane. |
| *Nef_polyhedron_2<T>::Aff_transformation* | affine transformations of the plane. |
| *Nef_polyhedron_2<T>::Boundary* { EXCLUDED, INCLUDED } | construction selection. |
| *Nef_polyhedron_2<T>::Content* { EMPTY, COMPLETE } | construction selection |

**3. Creation**

*Nef_polyhedron_2<T> N*(*Content plane = EMPTY*);

creates an instance $N$ of type *Nef_polyhedron_2<T>* and initializes it to the empty set if *plane == EMPTY* and to the whole plane if *plane == COMPLETE*.

*Nef_polyhedron_2<T> N*(*Line l*, *Boundary line = INCLUDED*);

creates a Nef polyhedron $N$ containing the halfplane left of $l$ including $l$ if *line == INCLUDED*, excluding $l$ if *line == EXCLUDED*.

template <*class Forward_iterator*>
*Nef_polyhedron_2<T> N*(*Forward_iterator it*, *Forward_iterator end*, *Boundary b = INCLUDED*);

creates a Nef polyhedron $N$ from the simple polygon $P$ spanned by the list of points in the iterator range $[it, end)$ and including its boundary if $b$ = *INCLUDED* and excluding the boundary otherwise. *Forward_iterator* has to be an iterator with value type *Point*. This construction expects that $P$ is simple. The degenerate cases where $P$ contains no point, one point or spans just one segment (two points) are correctly handled. In all degenerate cases there's only one unbounded face adjacent to the degenerate polygon. If *b == INCLUDED* then $N$ is just the boundary. If *b == EXCLUDED* then $N$ is the whole plane without the boundary.

**4. Operations**

| | | |
|---|---|---|
| *void* | *N*.clear(*Content plane = EMPTY*) | makes $N$ the empty set if *plane == EMPTY* and the full plane if *plane == COMPLETE*. |
| *bool* | *N*.is_empty( ) | returns true if $N$ is empty, false otherwise. |
| *bool* | *N*.is_plane( ) | returns true if $N$ is the whole plane, false otherwise. |

**Constructive Operations**

| | | |
|---|---|---|
| *Nef_polyhedron_2<T>* | *N*.complement( ) | returns the complement of *N* in the plane. |
| *Nef_polyhedron_2<T>* | *N*.interior( ) | returns the interior of *N*. |
| *Nef_polyhedron_2<T>* | *N*.closure( ) | returns the closure of *N*. |
| *Nef_polyhedron_2<T>* | *N*.boundary( ) | returns the boundary of *N*. |
| *Nef_polyhedron_2<T>* | *N*.regularization( ) | returns the regularized polyhedron (closure of interior). |
| *Nef_polyhedron_2<T>* | *N*.intersection(*Nef_polyhedron_2<T> N1*) | returns $N \cap N1$. |
| *Nef_polyhedron_2<T>* | *N*.join(*Nef_polyhedron_2<T> N1*) | returns $N \cup N1$. |
| *Nef_polyhedron_2<T>* | *N*.difference(*Nef_polyhedron_2<T> N1*) | returns $N - N1$. |
| *Nef_polyhedron_2<T>* | *N*.symmetric_difference(*Nef_polyhedron_2<T> N1*) | returns the symmectric difference $N - T \cup T - N$. |
| *Nef_polyhedron_2<T>* | *N*.transform(*Aff_transformation t*) | returns $t(N)$. |

Additionally there are operators $*, +, -, \hat{}, !$ which implement the binary operations *intersection*, *union*, *difference*, *symmetric difference*, and the unary operation *complement* respectively. There are also the corresponding modifi cation operations $*=, +=, -=, \hat{}=$.

There are also comparison operations like $<, \leq, >, \geq, ==, !=$ which implement the relations subset, subset or equal, superset, superset or equal, equality, inequality, respectively.

**Exploration - Point location - Ray shooting**

As Nef polyhedra are the result of forming complements and intersections starting from a set $H$ of halfspaces that are defi ned by oriented lines in the plane, they can be represented by an attributed plane map $M = (V, E, F)$. For topological queries within $M$ the following types and operations allow exploration access to this structure.

**5. Types**

*Nef_polyhedron_2<T>*::*Explorer*

a decorator to examine the underlying plane map. See the manual page of *Explorer*.

*Nef_polyhedron_2<T>*::*Object_handle*

a generic handle to an object of the underlying plane map. The kind of object (*vertex*, *halfedge*, *face*) can be determined and the object can be assigned to a corresponding handle by the three functions:
*bool assign*(*Vertex_const_handle*& *h*, *Object_handle*)
*bool assign*(*Halfedge_const_handle*& *h*, *Object_handle*)
*bool assign*(*Face_const_handle*& *h*, *Object_handle*)
where each function returns *true* iff the assignment to *h* was done.

*Nef_polyhedron_2<T>*::*Location_mode* { DEFAULT, NAIVE, LMWT }

selection flag for the point location mode.

**6. Operations**

| | |
|---|---|
| *bool* | *N*.contains(*Object_handle h*) |
| | returns true iff the object *h* is contained in the set represented by *N*. |
| *bool* | *N*.contained_in_boundary(*Object_handle h*) |
| | returns true iff the object *h* is contained in the 1-skeleton of *N*. |

*Object_handle* *N*.locate(*Point p*, *Location_mode m* = *DEFAULT*)

returns a generic handle *h* to an object (face, halfedge, vertex) of the underlying plane map that contains the point *p* in its relative interior. The point *p* is contained in the set represented by *N* if *N.contains*(*h*) is true. The location mode flag *m* allows one to choose between different point location strategies.

*Object_handle* *N*.ray_shoot(*Point p*, *Direction d*, *Location_mode m* = *DEFAULT*)

returns a handle *h* with *N.contains*(*h*) that can be converted to a *Vertex_/Halfedge_/Face_const_handle* as described above. The object returned is intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any object *h* of *N* with *N.contains*(*h*). The location mode flag *m* allows one to choose between different point location strategies.

*Object_handle* *N*.ray_shoot_to_boundary(*Point p*, *Direction d*, *Location_mode m* = *DEFAULT*)

returns a handle *h* that can be converted to a *Vertex_/Halfedge_const_handle* as described above. The object returned is part of the 1-skeleton of *N*, intersected by the ray starting in *p* with direction *d* and has minimal distance to *p*. The operation returns the null handle *NULL* if the ray shoot along *d* does not hit any 1-skeleton object *h* of *N*. The location mode flag *m* allows one to choose between different point location strategies.

*Explorer* *N*.explorer()

returns a decorator object which allows read-only access of the underlying plane map. See the manual page *Explorer* for its usage.

**Input and Output**

A Nef polyhedron *N* can be visualized in a *Window_stream W*. The output operator is defined in the file *CGAL/IO/Nef_polyhedron_2_Window_stream.h*.

**7. Implementation**

Nef polyhedra are implemented on top of a halfedge data structure and use linear space in the number of vertices, edges and facets. Operations like *empty* take constant time. The operations *clear*, *complement*, *interior*, *closure*, *boundary*, *regularization*, input and output take linear time. All binary set operations and comparison operations take time $O(n \log n)$ where $n$ is the size of the output plus the size of the input.

The point location and ray shooting operations are implemented in two flavors. The *NAIVE* operations run in linear query time without any preprocessing, the *DEFAULT* operations (equals *LMWT*) run in sub-linear query time, but preprocessing is triggered with the first operation. Preprocessing takes time $O(N^2)$, the sub-linear point location time is either logarithmic when LEDA's persistent dictionaries are present or if not then the point location time is worst-case linear, but experiments show often sublinear runtimes. Ray shooting equals point location plus a walk in the constrained triangulation overlayed on the plane map representation. The cost of the walk is proportional to the number of triangles passed in direction *d* until an obstacle is met. In a minimum weight triangulation of the obstacles (the plane map representing the polyhedron) the theory provides a $O(\sqrt{n})$ bound for the number of steps. Our locally minimum weight triangulation approximates the minimum weight triangulation only heuristically (the calculation of the minimum weight triangulation is conjectured to be NP hard). Thus we have no runtime guarantee but a strong experimental motivation for its approximation.

**8. Example**

Nef polyhedra are parameterized by a so-called extended geometric kernel. There are three kernels, one based on a homogeneous representation of extended points called *Extended_homogeneous*<*RT*> where *RT* is a ring type providing additionally a *gcd* operation and one based on a cartesian representation of extended points called *Extended_cartesian*<*NT*> where *NT* is a field type, and finally *Filtered_extended_homogeneous*<*RT*> (an optimized version of the first).

The member types of *Nef_polyhedron_2*< *Extended_homogeneous*<*NT*> > map to corresponding types of the CGAL geometry kernel (e.g. *Nef_polyhedron*::*Line* equals *CGAL*::*Homogeneous*<*leda_integer*>::*Line_2* in the example below).

```
#include <CGAL/basic.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Extended_homogeneous.h>
#include <CGAL/Nef_polyhedron_2.h>

using namespace CGAL;
typedef  Extended_homogeneous<leda_integer> Extended_kernel;
typedef  Nef_polyhedron_2<Extended_kernel>  Nef_polyhedron;
typedef  Nef_polyhedron::Line               Line;

int main()
{
  Nef_polyhedron N1(Line(1,0,0));
  Nef_polyhedron N2(Line(0,1,0), Nef_polyhedron::EXCLUDED);
  Nef_polyhedron N3 = N1 * N2; // line (*)
  return 0;
}
```

After line (*) *N3* is the intersection of *N1* and *N2*.

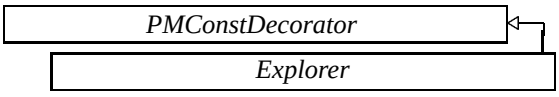
### 4.2.2 Plane map exploration ( Explorer )

**1. Definition**

An instance *E* of the data type *Explorer* is a decorator to explore the structure of the plane map underlying the Nef polyhedron. It inherits all topological adjacency exploration operations from *PMConstDecorator*. *Explorer* additionally allows one to explore the geometric embedding.

The position of each vertex is given by a so-called extended point, which is either a standard affine point or the tip of a ray touching an infinimaximal square frame centered at the origin. A vertex *v* is called a *standard* vertex if its embedding is a *standard* point and *non-standard* if its embedding is a *non-standard* point. By the straightline embedding of their source and target vertices, edges correspond to either affine segments, rays or lines or are part of the bounding frame.

**2. Generalization**




**3. Types**

*Explorer*::*Topological_explorer*
: The base class.

*Explorer*::*Point*
: the point type of finite vertices.

*Explorer*::*Ray*
: the ray type of vertices on the frame.

Iterators, handles, and circulators are inherited from *Topological_explorer*.

**4. Creation**

*Explorer* is copy constructable and assignable. An object can be obtained via the *Nef_polyhedron_2*::*explorer*( ) method of *Nef_polyhedron_2*.

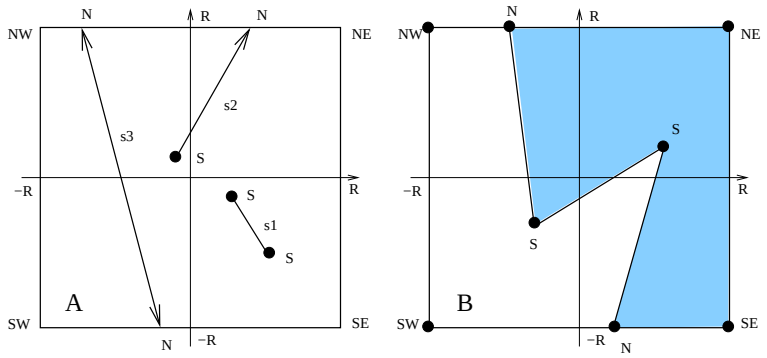



Figure 4.18: Extended geometry: standard vertices are marked by S, non-standard vertices are marked by N. **A**: The possible embeddings of edges: an affine segment s1, an affine ray s2, an affine line s3. **B**: A plane map embedded by extended geometry: note that the frame is arbitrarily large, the 6 vertices on the frame are at infinity, the two faces represent a geometrically unbounded area, however they are topologically closed by the frame edges. No standard point can be placed outside the frame.

**5. Operations**

*bool* *E*.is_standard(*Vertex_const_handle v*)

returns true iff *v*'s position is a standard point.

*Point* *E*.point(*Vertex_const_handle v*)

returns the standard point that is the embedding of *v*. *Precondition*: $E.is\_standard(v)$.

*Ray* *E*.ray(*Vertex_const_handle v*)

returns the ray defining the non-standard point on the frame. *Precondition*: $!E.is\_standard(v)$.

*bool* *E*.is_frame_edge(*Halfedge_const_handle e*)

returns true iff *e* is part of the infinimaximal frame.

### 4.2.3 Topological plane map exploration ( PMConstDecorator )

**1. Definition**

An instance *D* of the data type *PMConstDecorator* is a decorator for interfacing the topological structure of a plane map *P* (read-only).

A plane map *P* consists of a triple $(V,E,F)$ of vertices, edges, and faces. We collectively call them objects. An edge *e* is a pair of vertices $(v,w)$ with incidence operations $v = source(e)$, $w = target(e)$. The list of all edges with source *v* is called the adjacency list $A(v)$.

Edges are paired into twins. For each edge $e = (v,w)$ there's an edge $twin(e) = (w,v)$ and $twin(twin(e)) == e$[22].

An edge $e = (v,w)$ knows two adjacent edges $en = next(e)$ and $ep = previous(e)$ where $source(en) = w$, $previous(en) = e$ and $target(ep) = v$ and $next(ep) = e$. By this symmetric $previous - next$ relationship all edges are partitioned into face cycles. Two edges *e* and $e'$ are in the same face cycle if $e = next^*(e')$. All edges *e*

[22]The existence of the edge pairs makes *P* a bidirected graph, the *twin* links make *P* a map.

in the same face cycle have the same incident face $f = face(e)$. The cyclic order on the adjacency list of a vertex $v = source(e)$ is given by $cyclic\_adj\_succ(e) = twin(previous(e))$ and $cyclic\_adj\_pred(e) = next(twin(e))$.

A vertex $v$ is embedded via coordinates $point(v)$. By the embedding of its source and target an edge corresponds to a segment. $P$ has the property that the embedding is always *order-preserving*. This means a ray fixed in $point(v)$ of a vertex $v$ and swept around counterclockwise meets the embeddings of $target(e)$ ($e \in A(v)$) in the cyclic order defined by the list order of $A$.

The embedded face cycles partition the plane into maximal connected subsets of points. Each such set corresponds to a face. A face is bounded by its incident face cycles. For all the edges in the non-trivial face cycles it holds that the face is left of the edges. There can also be trivial face cycles in form of isolated vertices in the interior of a face. Each such vertex $v$ knows its surrounding face $f = face(v)$.

We call the embedded map $(V,E)$ also the 1-skeleton of $P$.

Plane maps are attributed. To each object $u \in V \cup E \cup F$ we attribute a value $mark(u)$ of type *Mark*. *Mark* fits the concepts assignable, default-constructible, and equal-comparable.

**2. Types**

| | |
|---|---|
| *PMConstDecorator*::*Plane_map* | The underlying plane map type |
| *PMConstDecorator*::*Point* | The point type of vertices. |
| *PMConstDecorator*::*Mark* | All objects (vertices, edges, faces) are attributed by a *Mark* object. |
| *PMConstDecorator*::*Size_type* | The size type. |

Local types are handles, iterators and circulators of the following kind: *Vertex_const_handle*, *Vertex_const_iterator*, *Halfedge_const_handle*, *Halfedge_const_iterator*, *Face_const_handle*, *Face_const_iterator*. Additionally the following circulators are defined.

*PMConstDecorator*::*Halfedge_around_vertex_const_circulator*
circulating the outgoing halfedges in $A(v)$.

*PMConstDecorator*::*Halfedge_around_face_const_circulator*
circulating the halfedges in the face cycle of a face $f$.

*PMConstDecorator*::*Hole_const_iterator* iterating all holes of a face $f$. The type is convertible to *Halfedge_const_handle*.

*PMConstDecorator*::*Isolated_vertex_const_iterator*
iterating all isolated vertices of a face $f$. The type generalizes *Vertex_const_handle*.

**3. Creation**

*PMConstDecorator* *D*(*const Plane_map*& *P*);
constructs a plane map decorator exploring $P$.

**4. Operations**

| | |
|---|---|
| *Vertex_const_handle* | *D*.source(*Halfedge_const_handle e*)<br>returns the source of $e$. |
| *Vertex_const_handle* | *D*.target(*Halfedge_const_handle e*)<br>returns the target of $e$. |
| *Halfedge_const_handle* | *D*.twin(*Halfedge_const_handle e*)<br>returns the twin of $e$. |
| *bool* | *D*.is_isolated(*Vertex_const_handle v*)<br>returns *true* iff $A(v) = \emptyset$. |

*Halfedge_const_handle* *D.*first_out_edge(*Vertex_const_handle v*)

returns one halfedge with source *v*. It's the starting point for the circular iteration over the halfedges with source *v*. *Precondition*: !*is_isolated*(*v*).

*Halfedge_const_handle* *D.*last_out_edge(*Vertex_const_handle v*)

returns the halfedge with source *v* that is the last in the circular iteration before encountering *first_out_edge*(*v*) again. *Precondition*: !*is_isolated*(*v*).

*Halfedge_const_handle* *D.*cyclic_adj_succ(*Halfedge_const_handle e*)

returns the edge after *e* in the cyclic ordered adjacency list of *source*(*e*).

*Halfedge_const_handle* *D.*cyclic_adj_pred(*Halfedge_const_handle e*)

returns the edge before *e* in the cyclic ordered adjacency list of *source*(*e*).

*Halfedge_const_handle* *D.*next(*Halfedge_const_handle e*)

returns the next edge in the face cycle containing *e*.

*Halfedge_const_handle* *D.*previous(*Halfedge_const_handle e*)

returns the previous edge in the face cycle containing *e*.

*Face_const_handle* *D.*face(*Halfedge_const_handle e*)

returns the face incident to *e*.

*Face_const_handle* *D.*face(*Vertex_const_handle v*)

returns the face incident to *v*. *Precondition*: *is_isolated*(*v*).

*Halfedge_const_handle* *D.*halfedge(*Face_const_handle f*)

returns a halfedge in the bounding face cycle of *f* (*Halfedge_const_handle*( ) if there is no bounding face cycle).

**Iteration**

*Halfedge_around_vertex_const_circulator* *D.*out_edges(*Vertex_const_handle v*)

returns a circulator for the cyclic adjacency list of *v*.

*Halfedge_around_face_const_circulator* *D.*face_cycle(*Face_const_handle f*)

returns a circulator for the outer face cycle of *f*.

*Hole_const_iterator* *D.*holes_begin(*Face_const_handle f*)

returns an iterator for all holes in the interior of *f*. A *Hole_iterator* can be assigned to a *Halfedge_around_face_const_circulator*.

*Hole_const_iterator* *D.*holes_end(*Face_const_handle f*)

returns the past-the-end iterator of *f*.

*Isolated_vertex_const_iterator* *D.*isolated_vertices_begin(*Face_const_handle f*)

returns an iterator for all isolated vertices in the interior of *f*.

*Isolated_vertex_const_iterator* *D.*isolated_vertices_end(*Face_const_handle f*)

returns the past the end iterator of *f*.

**Associated Information**

The type *Mark* is the general attribute of an object. The type *GenPtr* is equal to type *void*∗.

*const Point*& *D*.point(*Vertex_const_handle v*)
returns the embedding of *v*.

*const Mark*& *D*.mark(*Vertex_const_handle v*)
returns the mark of *v*.

*const Mark*& *D*.mark(*Halfedge_const_handle e*)
returns the mark of *e*.

*const Mark*& *D*.mark(*Face_const_handle f*)
returns the mark of *f*.

*const GenPtr*& *D*.info(*Vertex_const_handle v*)
returns a generic information slot.

*const GenPtr*& *D*.info(*Halfedge_const_handle e*)
returns a generic information slot.

*const GenPtr*& *D*.info(*Face_const_handle f*)
returns a generic information slot.

**Statistics and Integrity**

| | | |
|---|---|---|
| *Size_type* | *D*.number_of_vertices( ) | returns the number of vertices. |
| *Size_type* | *D*.number_of_halfedges( ) | returns the number of halfedges. |
| *Size_type* | *D*.number_of_edges( ) | returns the number of halfedge pairs. |
| *Size_type* | *D*.number_of_faces( ) | returns the number of faces. |
| *Size_type* | *D*.number_of_face_cycles( ) | returns the number of face cycles. |

*Size_type* *D*.number_of_connected_components( )
calculates the number of connected components of *P*.

*void* *D*.print_statistics(*std*::*ostream*& *os* = *std*::*cout*)
print the statistics of *P*: the number of vertices, edges, and faces.

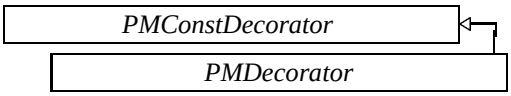
*void* *D*.check_integrity_and_topological_planarity(*bool faces* = *true*)
checks the link structure and the genus of *P*.

### 4.2.4 Plane map manipulation ( PMDecorator )

**1. Definition**

An instance *D* of the data type *PMDecorator* is a decorator to examine and modify a plane map. *D* inherits from *PMConstDecorator* but provides additional manipulation operations.

**2. Generalization**




**3. Types**

Local types are handles, iterators and circulators of the following kind: *Vertex_handle*, *Vertex_iterator*, *Halfedge_handle*, *Halfedge_iterator*, *Face_handle*, *Face_iterator*. Additionally the following circulators are defined. The *circulators* can be constructed from the corresponding halfedge handles or iterators.

*PMDecorator*::*Halfedge_around_vertex_circulator*

circulating the outgoing halfedges in $A(v)$.

*PMDecorator*::*Halfedge_around_face_circulator*

circulating the halfedges in the face cycle of a face $f$.

*PMDecorator*::*Hole_iterator*

iterating all holes of a face $f$. The type is convertible to *Halfedge_handle*.

*PMDecorator*::*Isolated_vertex_iterator*

iterating all isolated vertices of a face $f$. The type generalizes *Vertex_handle*.

*PMDecorator*:: { BEFORE, AFTER }

insertion order labels.

**4. Creation**

*PMDecorator D*(*Plane_map*& *p*);

constructs a decorator working on *P*.

**5. Operations**

| | | |
|---|---|---|
| *Plane_map*& | *D*.plane_map( ) | returns the plane map decorated. |
| *void* | *D*.clear( ) | reinitializes *P* to the empty map. |
| *Vertex_handle* | *D*.source(*Halfedge_handle e*) | returns the source of *e*. |
| *Vertex_handle* | *D*.target(*Halfedge_handle e*) | returns the target of *e*. |
| *Halfedge_handle* | *D*.twin(*Halfedge_handle e*) | returns the twin of *e*. |
| *bool* | *D*.is_isolated(*Vertex_handle v*) | returns *true* iff *v* is linked to the interior of a face. This is equivalent to the condition that $A(v) = \emptyset$. |
| *bool* | *D*.is_closed_at_source(*Halfedge_handle e*) | returns *true* when $prev(e) == twin(e)$. |
| *Halfedge_handle* | *D*.first_out_edge(*Vertex_handle v*) | returns a halfedge with source *v*. It's the starting point for the circular iteration over the halfedges with source *v*. *Precondition*: $!is\_isolated(v)$. |
| *Halfedge_handle* | *D*.last_out_edge(*Vertex_handle v*) | returns a the halfedge with source *v* that is the last in the circular iteration before encountering $first\_out\_edge(v)$ again. *Precondition*: $!is\_isolated(v)$. |
| *Halfedge_handle* | *D*.cyclic_adj_succ(*Halfedge_handle e*) | returns the edge after *e* in the cyclic ordered adjacency list of $source(e)$. |
| *Halfedge_handle* | *D*.cyclic_adj_pred(*Halfedge_handle e*) | returns the edge before *e* in the cyclic ordered adjacency list of $source(e)$. |
| *Halfedge_handle* | *D*.next(*Halfedge_handle e*) | returns the next edge in the face cycle containing *e*. |

*Halfedge_handle* *D*.previous(*Halfedge_handle e*)

returns the previous edge in the face cycle containing *e*.

*Face_handle* *D*.face(*Halfedge_handle e*)

returns the face incident to *e*.

*Face_handle* *D*.face(*Vertex_handle v*)

returns the face incident to *v*. *Precondition*: *is_isolated*$(v)$.

*Halfedge_handle* *D*.halfedge(*Face_handle* $f$)

returns a halfedge in the bounding face cycle of $f$ (*Halfedge_handle*( ) if there is no bounding face cycle).

**Iteration**

*Halfedge_around_vertex_circulator* *D*.out_edges(*Vertex_handle v*)

returns a circulator for the cyclic adjacency list of *v*.

*Halfedge_around_face_circulator* *D*.face_cycle(*Face_handle* $f$)

returns a circulator for the outer face cycle of $f$.

*Hole_iterator* *D*.holes_begin(*Face_handle* $f$)

returns an iterator for all holes in the interior of $f$. A *Hole_iterator* can be assigned to a *Halfedge_around_face_circulator*.

*Hole_iterator* *D*.holes_end(*Face_handle* $f$)

returns the past-the-end iterator of $f$.

*Isolated_vertex_iterator* *D*.isolated_vertices_begin(*Face_handle* $f$)

returns an iterator for all isolated vertices in the interior of $f$.

*Isolated_vertex_iterator* *D*.isolated_vertices_end(*Face_handle* $f$)

returns the past the end iterator of $f$.

**Update Operations**

*Vertex_handle* *D*.new_vertex(*Vertex_base vb* $=$ *Vertex_base*( ))

creates a new vertex.

*Face_handle* *D*.new_face(*Face_base fb* $=$ *Face_base*( ))

creates a new face.

*void* *D*.link_as_outer_face_cycle(*Face_handle* $f$, *Halfedge_handle e*)

makes *e* the entry point of the outer face cycle of $f$ and makes $f$ the face of all halfedges in the face cycle of *e*.

*void* *D*.link_as_hole(*Face_handle* $f$, *Halfedge_handle e*)

makes *e* the entry point of a hole face cycle of $f$ and makes $f$ the face of all halfedges in the face cycle of *e*.

*void* *D*.link_as_isolated_vertex(*Face_handle* $f$, *Vertex_handle v*)

makes *v* an isolated vertex within $f$.

*void* *D*.clear_face_cycle_entries(*Face_handle* $f$)

removes all isolated vertices and halfedges that are entrie points into face cycles from the lists of $f$.

*Halfedge_handle* *D*.new_halfedge_pair(*Vertex_handle v1*, *Vertex_handle v2*, *Halfedge_base hb* $=$ *Halfedge_base*( ))

creates a new pair of edges $(e1, e2)$ representing $(v1, v2)$ by appending the *ei* to $A(vi)$ $(i = 1, 2)$.

*Halfedge_handle* *D*.new_halfedge_pair(*Halfedge_handle e1*, *Halfedge_handle e2*, *Halfedge_base hb* = *Halfedge_base*( ), *int pos1* = *AFTER*, *int pos2* = *AFTER*)

creates a new pair of edges $(h1,h2)$ representing $(source(e1),source(e2))$ by inserting the *hi* before or after *ei* into the cyclic adjacency list of $source(ei)$ depending on *posi* $(i = 1,2)$ from *PMDecorator*::*BEFORE*, *PMDecorator*::*AFTER*.

*Halfedge_handle* *D*.new_halfedge_pair(*Halfedge_handle e*, *Vertex_handle v*, *Halfedge_base hb* = *Halfedge_base*( ), *int pos* = *AFTER*)

creates a new pair of edges $(e1,e2)$ representing $(source(e),v)$ by inserting *e1* before or after *e* into the cyclic adjacency list of $source(e)$ depending on *pos* from *PMDecorator*:: *BEFORE*, *PMDecorator*::*AFTER* and appending *e2* to $A(v)$.

*Halfedge_handle* *D*.new_halfedge_pair(*Vertex_handle v*, *Halfedge_handle e*, *Halfedge_base hb* = *Halfedge_base*( ), *int pos* = *AFTER*)

symmetric to the previous one.

*void* *D*.delete_halfedge_pair(*Halfedge_handle e*)

deletes *e* and its twin and updates the adjacency at its source and its target.

*void* *D*.delete_vertex(*Vertex_handle v*)

deletes *v* and all outgoing edges $A(v)$ as well as their twins. Updates the adjacency at the targets of the edges in $A(v)$.

*void* *D*.delete_face(*Face_handle f*)

deletes the face *f* without consideration of topological linkage.

*bool* *D*.has_outdeg_two(*Vertex_handle v*)

return true when *v* has outdegree two.

*void* *D*.merge_halfedge_pairs_at_target(*Halfedge_handle e*)

merges the halfedge pairs at $v = target(e)$. *e* and $twin(e)$ are preserved, $next(e)$, $twin(next(e))$ and *v* are deleted in the merger. *Precondition*: *v* has outdegree two. The adjacency at $source(e)$ and $target(next(e))$ is kept consistent.

**Incomplete topological update primitives**

*Halfedge_handle* *D*.new_halfedge_pair_at_source(*Vertex_handle v*, *int pos* = *AFTER*, *Halfedge_base hb* = *Halfedge_base*( ))

creates a new pair of edges $(e1,e2)$ representing $(v,(\,))$ by inserting *e1* at the beginning (BEFORE) or end (AFTER) of adjacency list of *v*.

*void* *D*.delete_halfedge_pair_at_source(*Halfedge_handle e*)

deletes *e* and its twin and updates the adjacency at its source.

*void* *D*.link_as_target_and_append(*Vertex_handle v*, *Halfedge_handle e*)

makes *v* the target of *e* and appends $twin(e)$ to $A(v)$.

*Halfedge_handle* *D*.new_halfedge_pair_without_vertices( )

inserts an open edge pair, and inits all link slots to their default handles.

*void* *D*.delete_vertex_only(*Vertex_handle v*)

deletes *v* without consideration of adjacency.

*void* *D*.delete_halfedge_pair_only(*Halfedge_handle e*)

deletes *e* and its twin without consideration of adjacency.

*void* *D*.link_as_target_of(*Halfedge_handle e*, *Vertex_handle v*)

makes $target(e) = v$ and sets *e* as the first in-edge if *v* was isolated before.

*void* *D*.link_as_source_of(*Halfedge_handle e*, *Vertex_handle v*)

makes $source(e) = v$ and sets *e* as the first out-edge if *v* was isolated before.

*void* *D*.make_first_out_edge(*Halfedge_handle e*)

makes *e* the first outgoing halfedge in the cyclic adjacency list of $source(e)$.

*void* *D*.set_adjacency_at_source_between(*Halfedge_handle e*, *Halfedge_handle en*)

makes *e* and *en* neigbors in the cyclic ordered adjacency list around $v = source(e)$. *Precondition*: $source(e) == source(en)$.

*void* *D*.set_adjacency_at_source_between(*Halfedge_handle e1*, *Halfedge_handle e_between*, *Halfedge_handle e2*)

inserts *e_between* into the adjacency list around $source(e1)$ between *e1* and *e2* and makes $source(e1)$ the source of *e_between*. *Precondition*: $source(e1) == source(e2)$.

*void* *D*.close_tip_at_target(*Halfedge_handle e*, *Vertex_handle v*)

sets *v* as target of *e* and closes the tip by setting the corresponding pointers such that $prev(twin(e)) == e$ and $next(e) == twin(e)$.

*void* *D*.close_tip_at_source(*Halfedge_handle e*, *Vertex_handle v*)

sets *v* as source of *e* and closes the tip by setting the corresponding pointers such that $prev(e) == twin(e)$ and $next(twin(e)) == e$.

*void* *D*.remove_from_adj_list_at_source(*Halfedge_handle e*)

removes a halfedge pair $(e, twin(e)$ from the adjacency list of $source(e)$. Afterwards $next(prev(e)) == next(twin(e))$ and $first\_out\_edge(source(e))$ is valid if $degree(source(v)) > 1$ before the operation.

*void* *D*.unlink_as_hole(*Halfedge_handle e*)

removes *e*'s existence as an face cycle entry point of $face(e)$. Does not update the face links of the corresponding face cycle halfedges.

*void* *D*.unlink_as_isolated_vertex(*Vertex_handle v*)

removes *v*'s existence as an isolated vertex in $face(v)$. Does not update *v*'s face link.

*void* *D*.link_as_prev_next_pair(*Halfedge_handle e1*, *Halfedge_handle e2*)

makes *e1* and *e2* adjacent in the face cycle $\ldots - e1 - e2 - \ldots$. Afterwards $e1 = previous(e2)$ and $e2 = next(e1)$.

*void* *D*.set_face(*Halfedge_handle e*, *Face_handle f*)

makes $f$ the face of *e*.

*void* *D*.set_face(*Vertex_handle v*, *Face_handle f*)

makes $f$ the face of *v*.

*void* *D*.set_halfedge(*Face_handle f*, *Halfedge_handle e*)

makes *e* entry edge in the outer face cycle of $f$.

*void* *D*.set_hole(*Face_handle f*, *Halfedge_handle e*)
makes *e* entry edge in a hole face cycle of *f*.

*void* *D*.set_isolated_vertex(*Face_handle f*, *Vertex_handle v*)
makes *v* an isolated vertex of *f*.

**Cloning**

*void* *D*.clone(*Plane_map H*)
clones *H* into *P*. Afterwards *P* is a copy of *H*.
*Precondition*: *H.check_integrity_and_topological_planarity*( ) and *P* is empty.

template <*typename LINKDA*>
*void* *D*.clone_skeleton(*Plane_map H*, *LINKDA L*)
clones the skeleton of *H* into *P*. Afterwards *P* is a copy of *H*. The link data accessor allows to transfer information from the old to the new objects. It needs the function call operators:
*void operator*( )(*Vertex_handle vn*, *Vertex_const_handle vo*) *const*
*void operator*( )(*Halfedge_handle hn*, *Halfedge_const_handle ho*) *const*
where *vn*, *hn* are the cloned objects and *vo*, *ho* are the original objects.
*Precondition*: *H.check_integrity_and_topological_planarity*( ) and *P* is empty.

**Associated Information**

*Point*& *D*.point(*Vertex_handle v*)
returns the embedding of *v*.

*Mark*& *D*.mark(*Vertex_handle v*)
returns the mark of *v*.

*Mark*& *D*.mark(*Halfedge_handle e*)
returns the mark of *e*.

*Mark*& *D*.mark(*Face_handle f*)
returns the mark of *f*.

*GenPtr*& *D*.info(*Vertex_handle v*)
returns a generic information slot.

*GenPtr*& *D*.info(*Halfedge_handle e*)
returns a generic information slot.

*GenPtr*& *D*.info(*Face_handle f*)
returns a generic information slot.

### 4.2.5 Extended Kernel Traits ( ExtendedKernelTraits_2 )

**1. Definition**

*ExtendedKernelTraits_2* is a kernel concept providing extended geometry[23]. Let *K* be an instance of the data type *ExtendedKernelTraits_2*. The central notion of extended geomtry are extended points. An extended point represents either a standard affine point of the Cartesian plane or a non-standard point representing the equivalence class of rays where two rays are equivalent if one is contained in the other.

Let *R* be an infinimaximal number[24], *F* be the square box with corners $NW(-R,R)$, $NE(R,R)$, $SE(R,-R)$, and $SW(-R,-R)$. Let *p* be a non-standard point and let *r* be a ray defining it. If the frame *F* contains the source

[23] It is called extended geometry for simplicity, though it is not a real geometry in the classical sense.
[24] A finite but very large number.

point of $r$ then let $p(R)$ be the intersection of $r$ with the frame $F$, if $F$ does not contain the source of $r$ then $p(R)$ is undefined. For a standard point let $p(R)$ be equal to $p$ if $p$ is contained in the frame $F$ and let $p(R)$ be undefined otherwise. Clearly, for any standard or non-standard point $p$, $p(R)$ is defined for any sufficiently large $R$. Let $f$ be any function on standard points, say with $k$ arguments. We call $f$ *extensible* if for any $k$ points $p_1, \ldots, p_k$ the function value $f(p_1(R), \ldots, p_k(R))$ is constant for all sufficiently large $R$. We define this value as $f(p_1, \ldots, p_k)$. Predicates like lexicographic order of points, orientation, and incircle tests are extensible.

An extended segment is defined by two extended points such that it is either an affine segment, an affine ray, an affine line, or a segment that is part of the square box. Extended directions extend the affine notion of direction to extended objects.

This extended geometry concept serves two purposes. It offers functionality for changing between standard affine and extended geometry. At the same time it provides extensible geometric primitives on the extended geometric objects.

**2. Types**

**Affine kernel types**

| | |
|---|---|
| *ExtendedKernelTraits_2*::*Standard_kernel* | the standard affine kernel. |
| *ExtendedKernelTraits_2*::*Standard_RT* | the standard ring type. |
| *ExtendedKernelTraits_2*::*Standard_point_2* | standard points. |
| *ExtendedKernelTraits_2*::*Standard_segment_2* | standard segments. |
| *ExtendedKernelTraits_2*::*Standard_line_2* | standard oriented lines. |
| *ExtendedKernelTraits_2*::*Standard_direction_2* | standard directions. |
| *ExtendedKernelTraits_2*::*Standard_ray_2* | standard rays. |
| *ExtendedKernelTraits_2*::*Standard_aff_transformation_2* | standard affine transformations. |

**Extended kernel types**

| | |
|---|---|
| *ExtendedKernelTraits_2*::*RT* | the ring type of our extended kernel. |
| *ExtendedKernelTraits_2*::*Point_2* | extended points. |
| *ExtendedKernelTraits_2*::*Segment_2* | extended segments. |
| *ExtendedKernelTraits_2*::*Direction_2* | extended directions. |
| *ExtendedKernelTraits_2*:: *Point_type* { SWCORNER, LEFTFRAME, NWCORNER, BOTTOMFRAME, STANDARD, TOPFRAME, SECORNER, RIGHTFRAME, NECORNER } | a type descriptor for extended points. |

**3. Operations**

**Interfacing the affine kernel types**

| | |
|---|---|
| *Point_2* | *K*.construct_point(*Standard_point_2 p*) |
| | creates an extended point and initializes it to the standard point $p$. |
| *Point_2* | *K*.construct_point(*Standard_line_2 l*) |
| | creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line $l$. |
| *Point_2* | *K*.construct_point(*Standard_point_2 p1*, *Standard_point_2 p2*) |
| | creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line $l(p1, p2)$. |
| *Point_2* | *K*.construct_point(*Standard_point_2 p*, *Standard_direction_2 d*) |
| | creates an extended point and initializes it to the equivalence class of all the rays underlying the ray starting in $p$ in direction $d$. |

| | | |
|---|---|---|
| *Point_2* | *K*.construct_opposite_point(*Standard_line_2 l*) | creates an extended point and initializes it to the equivalence class of all the rays underlying the oriented line opposite to *l*. |
| *Point_type* | *K*.type(*Point_2 p*) | determines the type of *p* and returns it. |
| *bool* | *K*.is_standard(*Point_2 p*) | returns *true* iff *p* is a standard point. |
| *Standard_point_2* | *K*.standard_point(*Point_2 p*) | returns the standard point represented by *p*. *Precondition*: *K.is_standard*(*p*). |
| *Standard_line_2* | *K*.standard_line(*Point_2 p*) | returns the oriented line representing the bundle of rays defining *p*. *Precondition*: !*K.is_standard*(*p*). |
| *Standard_ray_2* | *K*.standard_ray(*Point_2 p*) | a ray defining *p*. *Precondition*: !*K.is_standard*(*p*). |
| *Point_2* | *K*.NE() | returns the point on the northeast frame corner. |
| *Point_2* | *K*.SE() | returns the point on the southeast frame corner. |
| *Point_2* | *K*.NW() | returns the point on the northwest frame corner. |
| *Point_2* | *K*.SW() | returns the point on the southwest frame corner. |

**Geometric kernel calls**

| | | |
|---|---|---|
| *Point_2* | *K*.source(*Segment_2 s*) | returns the source point of *s*. |
| *Point_2* | *K*.target(*Segment_2 s*) | returns the target point of *s*. |
| *Segment_2* | *K*.construct_segment(*Point_2 p*, *Point_2 q*) | constructs a segment *pq*. |
| *int* | *K*.orientation(*Segment_2 s*, *Point_2 p*) | returns the orientation of *p* with respect to the line through *s*. |
| *int* | *K*.orientation(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*) | returns the orientation of *p3* with respect to the line through *p1p2*. |
| *bool* | *K*.leftturn(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*) | return true iff the *p3* is left of the line through *p1p2*. |
| *bool* | *K*.is_degenerate(*Segment_2 s*) | return true iff *s* is degenerate. |
| *int* | *K*.compare_xy(*Point_2 p1*, *Point_2 p2*) | returns the lexicographic order of *p1* and *p2*. |
| *int* | *K*.compare_x(*Point_2 p1*, *Point_2 p2*) | returns the order on the *x*-coordinates of *p1* and *p2*. |
| *int* | *K*.compare_y(*Point_2 p1*, *Point_2 p2*) | returns the order on the *y*-coordinates of *p1* and *p2*. |
| *Point_2* | *K*.intersection(*Segment_2 s1*, *Segment_2 s2*) | returns the point of intersection of the lines supported by *s1* and *s2*. *Precondition*: the intersection point exists. |
| *Direction_2* | *K*.construct_direction(*Point_2 p1*, *Point_2 p2*) | returns the direction of the vector *p2 - p1*. |
| *bool* | *K*.strictly_ordered_ccw(*Direction_2 d1*, *Direction_2 d2*, *Direction_2 d3*) | returns *true* iff *d2* is in the interior of the counterclockwise angular sector between *d1* and *d3*. |

*bool* *K*.strictly_ordered_along_line(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*)
returns *true* iff *p2* is in the relative interior of the segment *p1p3*.

*bool* *K*.contains(*Segment_2 s*, *Point_2 p*)
returns true iff *s* contains *p*.

*bool* *K*.first_pair_closer_than_second(*Point_2 p1*, *Point_2 p2*, *Point_2 p3*, *Point_2 p4*)
returns true iff $|p1 - p2| < |p3 - p4|$.

*char** *K*.output_identifier() returns a unique identifier for kernel object input/output.

### 4.2.6 Polynomials in one variable ( RPolynomial )

**1. Definition**

An instance *p* of the data type *RPolynomial<NT>* represents a polynomial $p = a_0 + a_1x + \ldots a_dx^d$ from the ring $NT[x]$. The data type offers standard ring operations and a sign operation which determines the sign for the limit process $x \to \infty$.

$NT[x]$ becomes a unique factorization domain, if the number type *NT* is either a field type (1) or a unique factorization domain (2). In both cases there's a polynomial division operation defined.

**2. Types**

*RPolynomial<NT>::NT* the component type representing the coefficients.

*RPolynomial<NT>::const_iterator* a random access iterator for read-only access to the coefficient vector.

**3. Creation**

*RPolynomial<NT> p*;
introduces a variable *p* of type *RPolynomial<NT>* of undefined value.

*RPolynomial<NT> p*(*NT a0*);
introduces a variable *p* of type *RPolynomial<NT>* representing the constant polynomial $a_0$.

*RPolynomial<NT> p*(*NT a0*, *NT a1*);
introduces a variable *p* of type *RPolynomial<NT>* representing the polynomial $a_0 + a_1x$.

*RPolynomial<NT> p*(*NT a0*, *NT a1*, *NT a2*);
introduces a variable *p* of type *RPolynomial<NT>* representing the polynomial $a_0 + a_1x + a_2x^2$.

template <*class Forward_iterator*>
*RPolynomial<NT> p*(*Forward_iterator first*, *Forward_iterator last*);
introduces a variable *p* of type *RPolynomial<NT>* representing the polynomial whose coefficients are determined by the iterator range, i.e. let $(a_0 = *first, a_1 = *++first, \ldots a_d = *it)$, where $++it == last$ then *p* stores the polynomial $a_1 + a_2x + \ldots a_dx^d$.

**4. Operations**

| | | |
|---|---|---|
| *int* | *p*.degree() | the degree of the polynomial. |
| *const NT&* | *p*[*unsigned int i*] | the coefficient $a_i$ of the polynomial. |
| *const_iterator* | *p*.begin() | a random access iterator pointing to $a_0$. |
| *const_iterator* | *p*.end() | a random access iterator pointing beyond $a_d$. |

| | | |
|---|---|---|
| *NT* | *p*.eval_at(*NT R*) | evaluates the polynomial at *R*. |
| *CGAL*::*Sign* | *p*.sign() | returns the sign of the limit process for $x \to \infty$ (the sign of the leading coeffi cient). |
| *bool* | *p*.is_zero() | returns true iff *p* is the zero polynomial. |
| *RPolynomial*<*NT*> | *p*.abs() | returns $-p$ if $p.sign() == NEGATIVE$ and *p* otherwise. |
| *NT* | *p*.content() | returns the content of *p* (the gcd of its coeffi cients). *Precondition*: Requires *NT* to provide a *gdc* operation. |

Additionally *RPolynomial*<*NT*> offers standard arithmetic ring opertions like $+, -, *, +=, -=, *=$. By means of the sign operation we can also offer comparison predicates as $<, >, \leq, \geq$. Where $p_1 < p_2$ holds iff $sign(p_1 - p_2) < 0$. This data type is fully compliant to the requirements of CGAL number types.

*RPolynomial*<*NT*> *p1* / *p2*
: implements polynomial division of *p1* and *p2*. if $p1 = p2 * p3$ then *p2* is returned. The result is undefi ned if *p3* does not exist in $NT[x]$. The correct division algorithm is chosen according to a traits class *ring_or_fi eld*<*NT*> provided by the user. If $ring\_or\_field<NT>::kind == ring\_with\_gcd$ then the division is done by *pseudo division* based on a *gcd* operation of *NT*. If $ring\_or\_field<NT>::kind == field\_with\_div$ then the division is done by *euclidean division* based on the division operation of the fi eld *NT*.
**Note** that $NT = int$ quickly leads to overfbw errors when using this operation.

**Non member functions**

*RPolynomial*<*NT*> gcd(*RPolynomial*<*NT*> *p1*, *RPolynomial*<*NT*> *p2*)
: returns the greatest common divisor of *p1* and *p2*. **Note** that $NT = int$ quickly leads to overfbw errors when using this operation. *Precondition*: Requires *NT* to be a unique factorization domain, i.e. to provide a *gdc* operation.

*void* pseudo_div(*RPolynomial*<*NT*> *f*, *RPolynomial*<*NT*> *g*, *RPolynomial*<*NT*>& *q*, *RPolynomial*<*NT*>& *r*, *NT*& *D*)
: implements division with remainder on polynomials of the ring $NT[x]$: $D * f = g * q + r$. *Precondition*: *NT* is a unique factorization domain, i.e., there exists a *gcd* operation and an integral division operation on *NT*.

*void* euclidean_div(*RPolynomial*<*NT*> *f*, *RPolynomial*<*NT*> *g*, *RPolynomial*<*NT*>& *q*, *RPolynomial*<*NT*>& *r*)
: implements division with remainder on polynomials of the ring $NT[x]$: $f = g * q + r$. *Precondition*: *NT* is a fi eld, i.e., there exists a division operation on *NT*.

**5. Implementation**

This data type is implemented as an item type via a smart pointer scheme. The coeffi cients are stored in a vector of *NT* entries. The simple arithmetic operations $+, -$ take time $O(d * T(NT))$, multiplication is quadratic in maximal degree of the arguments times $T(NT)$, where $T(NT)$ is the time for a corresponding operation on two instances of the ring type.

**Range template**

template <*class Forward_iterator*>
*typename std*::*iterator_traits*<*Forward_iterator*>::*value_type*
gcd_of_range(*Forward_iterator its*, *Forward_iterator ite*)
: calculates the greates common divisor of the set of numbers $\{*its, *++its, \ldots, *it\}$ of type *NT*, where $++it == ite$ and *NT* is the value type of *Forward_iterator*. *Precondition*: there exists a pairwise gcd operation $NT\ gcd(NT, NT)$ and $its != ite$.

# Plane Map Overlay ( PM_overlayer )

**1. Definition**

An instance $O$ of data type *PM_overlayer*<*PMD*, *GEO*> is a decorator object offering plane map overlay calculation. Overlay is either calculated from two plane maps or from a set of segments. The result is stored in a plane map $P$ that carries the geometry and the topology of the overlay.

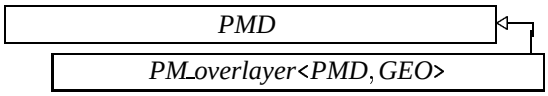
The two template parameters allow to adapt the overlay calculation to different scenarios. The template parameter *PMD* has to be a model conforming to our plane map decorator concept *PMDecorator*. The concept describes the interface how the topological information stored in $P$ can be extracted. The geometry *GEO* has to be a model conforming to the concept *OverlayerGEO2*.

The overlay of a set of segments $S$ is stored in a plane map $P = (V, E, F)$. Vertices are either the endpoints of segments (trivial segments are allowed) or the result of a non-degenerate internal intersection of two segments. Between two vertices there's an edge if there's a segment that supports the straight line embedding of $e$ and if there's no vertex in the relative interior of the embedding of $e$.

The faces refer to the maximal connected open point sets of the planar subdivision implied by the embedding of the vertices and edges. Faces are bounded by possibly several face cycles[25] including isolated vertices. The overlay process in the method *create* creates the objects, the topology of the result and allows to link the plane map objects to input segments by means of a data accessor. The method starts from zero- and one-dimensional geometric objects in $S$ and produces a plane map $P$ where each point of the plane can be assigned to an object (vertex, edge, or face) of $P$.

The overlay of two plane maps $P_i = (V_i, E_i, F_i)$ has the additional aspect that we already start from two planar subdivisions. We use the index $i = 0, 1$ defining the reference to $P_i$, unindexed variables refer to the resulting plane map $P$. The 1-skeleta of the two maps subdivide the edges and faces of the complementary structure into smaller units. This means vertices and edges of $P_i$ can split edges of $P_{1-i}$ and face cycles of $P_i$ subdivide faces of $P_{1-i}$. The 1-skeleton $P'$ of $P$ is defined by the overlay of the embedding of the 1-skeleta of $P_0$ and $P_1$ (Take a trivial segment for each vertex and a segment for each edge and use the overlay definition of a set of segments above). The faces of $P$ refer to the maximal connected open point sets of the planar subdivision implied by the embedding of $P'$. Each object from the output tuple $(V, E, F)$ has a *supporting* object $u_i$ in each of the two input structures. Imagine the two maps to be transparencies, which we stack. Then each point of the plane is covered by an object from each of the input structures. This support relation from the input structures to the output structure defines an information flow. Each supporting object $u_i$ of $u$ $(i = 0, 1)$ carries an attribute $mark(u_i)$. After the subdivision operation this attribute is associated to the output object $u$ by $mark(u, i)$.

**2. Generalization**




**3. Types**

| | |
|---|---|
| *PM_overlayer*<*PMD*, *GEO*>::*Decorator* | the plane map decorator *PMD*. |
| *PM_overlayer*<*PMD*, *GEO*>::*Plane_map* | the plane map type decorated by *PMD*. |
| *PM_overlayer*<*PMD*, *GEO*>::*Geometry* | the geometry kernel *GEO*. |
| *PM_overlayer*<*PMD*, *GEO*>::*Point* | the point type of the geometric kernel, *Precondition*: *Point* equals *Plane_map*::*Point*. |
| *PM_overlayer*<*PMD*, *GEO*>::*Segment* | the segment type of the geometric kernel. |
| *PM_overlayer*<*PMD*, *GEO*>::*Mark* | the attribute type of plane map objects. |

[25] For the definition of plane maps and their concepts see the manual page of *PMConstDecorator*.

**4. Creation**

*PM_overlayer*<*PMD*, *GEO*> *O*(*Plane_map*& *P*, *Geometry g* = *Geometry*( ));

*O* is a decorator object manipulating *P*.

**5. Operations**

template <*typename Forward_iterator*, *typename Object_data_accessor*>
*void* *O*.create(*Forward_iterator start*, *Forward_iterator end*, *Object_data_accessor*& *A*)

produces in *P* the plane map consistent with the overlay of the segments from the iterator range $[start, end)$. The data accessor *A* allows to initialize created vertices and edges with respect to the segments in the iterator range. *A* requires the following methods:

```
void supporting_segment(Halfedge_handle e, Forward_iterator it)
void trivial_segment(Vertex_handle v, Forward_iterator it)
void starting_segment(Vertex_handle v, Forward_iterator it)
void passing_segment(Vertex_handle v, Forward_iterator it)
void ending_segment(Vertex_handle v, Forward_iterator it)
```

where *supporting_segment* is called for each non-trivial segment $*it$ supporting a newly created edge *e*, *trivial_segment* is called for each trivial segment $*it$ supporting a newly created vertex *v*, and the three last operations are called for each non-trivial segment $*it$ starting at/passing through/ending at the embedding of a newly created vertex *v*. *Precondition*: *Forward_iterator* has value type *Segment*.

*void* *O*.subdivide(*Plane_map P0*, *Plane_map P1*)

constructs the overlay of the plane maps *P0* and *P1* in *P*, where all objects (vertices, halfedges, faces) of *P* are *enriched* by the marks of the supporting objects of the two input structures: e.g. let *v* be a vertex supported by a node *v0* in *P0* and by a face *f1* in *P1* and *D0*, *D1* be decorators of type *PM_decorator* on *P0*,*P1*. Then $O.mark(v,0) = D0.mark(v0)$ and $O.mark(v,1) = D1.mark(f1)$.

template <*typename Selection*>
*void* *O*.select(*Selection*& *predicate*)

sets the marks of all objects according to the selection predicate *predicate*. *Selection* has to be a function object type with a function operator

```
Mark operator()(Mark m0, Mark m1)
```

For each object *u* of *P* enriched by the marks of the supporting objects according to the previous procedure *subdivide*, after this operation $O.mark(u) = predicate\ (\ O.mark(u,0), O.mark(u,1)\ )$. The additional marks are invalidated afterwards.

template <*typename Keep_edge*>
*void* *O*.simplify(*Keep_edge keep*)

simplifies the structure of *P* according to the marks of its objects. An edge *e* separating two faces *f1* and *f2* and equal marks $mark(e) == mark(f1) == mark(f2)$ is removed and the faces are unified. An isolated vertex *v* in a face $f$ with $mark(v) == mark(f)$ is removed. A vertex *v* with outdegree two, two collinear out-edges *e1*,*e2* and equal marks $mark(v) == mark(e1) == mark(e2)$ is removed and the edges are unified. The data accessor *keep* requires the function call operator

```
bool operator()(Halfedge_handle e)
```

that allows to avoid the simplification for edge pairs referenced by *e*.

### 4.2.1 Output traits for segment overlay ( SegmentOverlayOutput )

**1. Definition**

This is the plane map decorator concept for the *PMDEC* template parameter of *PM_seg_overlay_traits*.

**2. Types**

*SegmentOverlayOutput*::*Vertex_handle*

the vertex handle.

*SegmentOverlayOutput*::*Halfedge_handle*

the halfedge handle.

*SegmentOverlayOutput*::*Point_2*

embedding type. *Precondition*: *Point_2* equals *GEOM*::*Point_2*.

**3. Creation**

Let *G* be an object of type *SegmentOverlayOutput.*

**4. Operations**

| | | |
|---|---|---|
| *Vertex_handle* | *G*.new_vertex(*Point_2 p*) | creates a new vertex in the output structure and embeds it via the point *p*. |
| *void* | *G*.link_as_target_and_append(*Vertex_handle v*, *Halfedge_handle e*) | makes *v* the target of *e* and appends the twin of *e* (its reversal edge) to *v*'s adjacency list. |
| *Halfedge_handle* | *G*.new_halfedge_pair_at_source(*Vertex_handle v*) | returns a newly created edge inserted before the first edge of the adjacency list of *v*. It also creates a reversal edge whose target is *v*. |

**Additional sweep information**

The iterator type *ITERATOR* has to be the same type as the first type parameter of *Segment_overlay_traits*.

| | | |
|---|---|---|
| *void* | *G*.supporting_segment(*Halfedge_handle e*, *ITERATOR it*) | the non-trivial segment $*it$ supports the edge *e*. |
| *void* | *G*.trivial_segment(*Vertex_handle v*, *ITERATOR it*) | the trivial segment $*it$ supports vertex *v*. |
| *void* | *G*.halfedge_below(*Vertex_handle v*, *Halfedge_handle e*) | associates the edge *e* as the edge below *v*. |
| *void* | *G*.starting_segment(*Vertex_handle v*, *ITERATOR it*) | the segment $*it$ starts in *v*. |
| *void* | *G*.passing_segment(*Vertex_handle v*, *ITERATOR it*) | the segment $*it$ passes *v* (contains it in its relative interior) . |
| *void* | *G*.ending_segment(*Vertex_handle v*, *ITERATOR it*) | the segment $*it$ ends in *v*. |

### 4.2.2 Geometry for segment overlay ( SegmentOverlayGeometry_2 )

**1. Definition**

*SegmentOverlayGeometry_2* is a kernel concept providing affine geometry for the overlay of segment. The concept specifies geometric types, predicates, and constructions.

**2. Types**

Local types are *Point_2*, *Segment_2*, the ring type *RT*, and the field type *FT*. See the CGAL 2d kernel for a description of *RT* and *FT*.

**3. Creation**

The kernel must be default and copy constructible. Let *K* be an object of type *SegmentOverlayGeometry_2*.

**4. Operations**

| | | |
|---|---|---|
| *Point_2* | *K*.source(*Segment_2 s*) | returns the source point of *s*. |
| *Point_2* | *K*.target(*Segment_2 s*) | returns the target point of *s*. |
| *bool* | *K*.is_degenerate(*Segment_2 s*) | return true iff *s* is degenerate. |
| *int* | *K*.compare_xy(*Point_2 p1*, *Point_2 p2*) | returns the lexicographic order of *p1* and *p2*. |
| *Segment_2* | *K*.construct_segment(*Point_2 p*, *Point_2 q*) | constructs a segment *pq*. |
| *int* | *K*.orientation(*Segment_2 s*, *Point_2 p*) | returns the orientation of *p* with respect to the line through *s*. |
| *Point_2* | *K*.intersection(*Segment_2 s1*, *Segment_2 s2*) | returns the point of intersection of the lines supported by *s1* and *s2*. The algorithm asserts that this intersection point exists. |

### 4.2.3 A Generic Plane Sweep Framework ( generic_sweep )

**1. Definition**

The data type *generic_sweep<T>* provides a general framework for algorithms following the plane sweep paradigm. The plane sweep paradigm can be described as follows. A vertical line sweeps the plane from left to right and constructs the desired output incrementally left behind the sweep line. The sweep line maintains knowledge of the scenery of geometric objects and stops at points where changes of this knowledge relevant for the output occur. These points are called events.

A general plane sweep framework structures the execution of the sweep into phases and provides a storage place for all data structures necessary to execute the sweep. An object *GS* of type *generic_sweep<T>* maintains an object of type *T* which generally can be used to store necessary structures. The content is totally dependent of the sweep specification and thereby varies within the application domain of the framework.

The traits class *T* has to provide a set of types which define the input/output interface of the sweep: the input type *INPUT*, the output type *OUTPUT*, and a geometry kernel type *GEOMETRY*.

The natural phases which determine a sweep are

```
// INITIALIZATION
initialize_structures();
check_invariants();

// SWEEP LOOP
while ( event_exists() ) {
  process_event();
  check_invariants();
  procede_to_next_event();
}

// COMPLETION
```

```
complete_structures();
check_final();
```

**Initialization** – initializing the data structures, ensuring preconditions, checking invariants

**Sweep Loop** – iterating over all events, while handling the event stops, ensuring invariants and the soundness of all data structures and maybe triggering some animation tasks.

**Completion** – cleaning up some data structures and completing the output.

The above subtasks are specified in the traits concept *GenericSweepTraits*.

**2. Types**

| | |
|---|---|
| *generic_sweep*<*T*>::*TRAITS* | the traits class |
| *generic_sweep*<*T*>::*INPUT* | the input interface. |
| *generic_sweep*<*T*>::*OUTPUT* | the output container. |
| *generic_sweep*<*T*>::*GEOMETRY* | the geometry kernel. |

**3. Creation**

*generic_sweep*<*T*> *PS*(*INPUT input*, *OUTPUT*& *output*, *GEOMETRY geometry* = *GEOMETRY*( ));

creates a plane sweep object for a sweep on objects determined by *input* and delivers the result of the sweep in *output*. The traits class *T* specifies the models of all types and the implementations of all methods used by *generic_sweep*<*T*>. At this point, it suffices to say that *INPUT* represents the input data type and *OUTPUT* represents the result data type. The *geometry* is an object providing object bound, geometry traits access.

*generic_sweep*<*T*> *PS*(*OUTPUT*& *output*, *GEOMETRY geometry* = *GEOMETRY*( ));

a simpler call of the above where *output* carries also the input.

**4. Operations**

| | | |
|---|---|---|
| *void* | *PS*.sweep() | execute the plane sweep. |

**5. Example**

A typical sweep based on *generic_sweep*<*T*> looks like the following little program:

```
typedef std::list<POINT>::const_iterator iterator;
typedef std::pair<iterator,iterator> iterator_pair;
std::list<POINT>  P; // fill input
GRAPH<POINT,LINE> G; // the output
generic_sweep<triang_sweep_traits>
  triangulation(iterator_pair(P.begin(),P.end()),G);
triangulation.sweep();
```

**6. Events**

To enable animation of the sweep there are event hooks inserted which allow an observer to attach certain visualization actions to them. There are four such hooks:

| | |
|---|---|
| *PS.post_init_hook*(*TRAITS*&) | triggered just after initialization. |
| *PS.pre_event_hook*(*TRAITS*&) | triggered just before the sweep event. |
| *PS.post_event_hook*(*TRAITS*&) | triggered just after the sweep event. |

*PS.post_completion_hook*(*TRAITS*&) triggered just after the completion phase.

All of these are triggered during the sweep with the instance of the *TRAITS* class that is stored inside the plane sweep object. Thus any animation operation attached to a hook can work on that class object which maintains the sweep status.

### 4.2.4 Traits concept for the generic sweep ( GenericSweepTraits )

**1. Definition**

*GenericSweepTraits* is the concept for the template parameter *T* of the generic plane sweep class *generic_sweep*<*T*>. It defines the interface that has to be implemented to adapt the generic sweep framework to a concrete instance.

**2. Types**

*GenericSweepTraits*::*INPUT* the input interface.

*GenericSweepTraits*::*OUTPUT* the output container.

*GenericSweepTraits*::*GEOMETRY*
the geometry used.

**3. Creation**

*GenericSweepTraits T*(*INPUT in*, *OUTPUT*& *out*, *GEOMETRY geom*);
creates an object *T* of type *GenericSweepTraits*, and allows thereby to transport the input/output data into the traits class.

*GenericSweepTraits T*(*OUTPUT*& *out*, *GEOMETRY geom*);
creates an object *T* of type *GenericSweepTraits*, and allows thereby to transport the input/output data into the traits class.

**4. Operations**

| | | |
|---|---|---|
| *void* | *T*.initialize_structures() | codes initialization of structures before the sweep loop. |
| *bool* | *T*.event_exists() | codes loop control at the beginning of the sweep loop body. |
| *void* | *T*.procede_to_next_event() | codes loop progress at the end of the sweep loop body. |
| *void* | *T*.process_event() | codes the actual event handling (the loop body). |
| *void* | *T*.complete_structures() | codes the completion phase after the sweep loop. |
| *void* | *T*.check_invariants() | allows checking sweep loop invariants. |
| *void* | *T*.check_final() | allows checking of final invariants (after completion). |

## 4.3 English Summary

This thesis presents research and software engineering in the field of computational geometry. Computational geometry is the scientific field of computer science that tackles geometric problems and provides efficient algorithms for their solution. Our aim is to provide algorithms and implementations thereof that are sound in theory and prove their efficiency in implementations. When implementing software, the last step of packaging and offering it to potential users in the outside world — whether the computer science community or experts from other sciences — is still a major task. This work tries to document the whole process: the theoretical basics, the software design, and the software as part of a software library project.

This thesis describes two software components and their underlying theory. Its results are mainly located in the domain of software engineering but we also present an extension of classical Euclidean geometry that was established in the research for the realization of Nef polyhedra. Our software modules are strongly anchored in and supported by the underlying theory.

The practical part of this work had an impact on and is based on the two software libraries LEDA and CGAL. Both are software libraries that offer solutions to problems in the domain of computational geometry. Where LEDA offers an easy but monolithic approach to its data structures and algorithms, more ambitious techniques are used in CGAL. CGAL is designed in the spirit of generic programming and in this sense is also a software engineering experiment that tries to exploit C++ template technology to the extreme. The use of templates allows pattern-based compile-time polymorphism as opposed to execution-time polymorphism via inheritance and virtual functions. Only template technology offers code composition at compile-time without runtime penalties.

The first part of the thesis describes a module offering higher-dimensional Euclidean geometry. It describes the objects and primitives that support the development of geometric algorithms in $d$-space. Our contribution is the design of the interface consisting of the objects together with predicates and constructions. Special care was taken to refine the concepts that allow generic adaptation of the kernel from the original monolithic design, *e.g.*, the number types and their docking into the kernel functionality via a linear algebra module. To enhance usability the representation-based kernel families can at the same time be used as traits classes in the instantiation of application classes. The traits classes here encapsulate the geometric primitives that control the logical flow of algorithms. This is one general design pattern of CGAL starting with version 2.3.

Our second project concerns the design and implementation of planar Nef polyhedra. The corresponding abstract definition of this polyhedral framework was founded by the Swiss mathematician W. Nef. Our design uses plane maps for the topological description (finite representation) of planar Nef polyhedra. To unify the treatment of the finite and infinite character of the vertices we use extended points as introduced in the third chapter of this thesis. The strength of this design is its clear separation of the special geometric demands of Nef polyhedra from the topological structure used to represent them. We can thereby show that standard affine plane map overlay as presented in the algorithmic part of the second chapter can be used transparently for the solution of the geometrically unbounded but symbolically bounded overlay problem that is part of the binary operations of Nef polyhedra.

We introduce the notion of infimaximal frames as an extension of affine geometry. Although it is a vital part in the realization of Nef polyhedra it has also further applications. We present the

theory and describe the implementation of three extended kernels as used for the Nef polyhedron software. We use simple but efficient algebraic techniques that aim for different strengths. Two kernels trade efficiency for simplicity. In these kernels, a polynomial ring type is used to realize the arithmetic operations that occur in the implementation of kernel predicates and constructions. Thereby the original geometric complexity with respect to the embedding of affine and frame-supported points is transferred into algebraic complexity that can easily be processed by the polynomial data type. The strength of this implementation is its verifiable correctness. The third kernel trades simplicity for efficiency. We use standard filtering techniques and unroll the occurring algebraic expressions explicitly to obtain a runtime optimized kernel.

Generic programming is programming with concepts and models where models are concrete realizations of the abstract concepts. Unfortunately the C++ language constructs and the corresponding compilers only weakly support the checking of whether models fit concepts during the instantiation process. Rendering generic software modules useful requires a major documentation effort. The reasonable design of the concepts, the correct implementation of their models, but also their expressive documentation are unavoidable requirements for good software design. This thesis describes all facets in a literate programming style. We give the abstract motivation for the design, show the techniques used to implement the modules and cite excerpts from the documentation in the appendix.

What is gained by genericity? Generic programming allows code extension and adaptation without cut-and-paste programming. Code can easily be tuned and adapted towards a user's needs. Generic composition supports the reuse of software in a well-defined way. Moreover, it allows experimental exchange of models and fast prototyping.

## 4.4 Deutsche Zusammenfassung

Diese Arbeit beschreibt Forschungsergebnisse und Softwareentwicklung im Gebiet der algorithmischen Geometrie. Die algorithmische Geometrie bildet ein Gebiet innerhalb der Informatik, das sich hauptsächlich mit der Lösung von geometrischen Problemen durch resourceneffiziente Algorithmen befasst. Die vollständige Umsetzung dieses Ansatzes erfordert Algorithmen, die theoretisch korrekt sind, und entsprechende Implementierungen, die auch im praktischen Laufzeitvergleich ihre Effizienz unter Beweis stellen. Wenn man diese Software einer breiteren Benutzergruppe zur Verfügung stellen will — aus der Informatik oder sogar anderen Wissenschaftsgebieten, verbleibt nach der Implementierung noch die Aufgabe, die Software entsprechend aufzubereiten und zu dokumentieren. Der letzte Schritt ist ein nicht-trivialer Teil der Ingenieuraufgabe. Wir dokumentieren hier den gesamten Prozess: die theoretischen Grundlagen, das Software-Design und die Software als Teil einer Softwarebibliothek.

In dieser Arbeit werden zwei Softwarekomponenten zusammen mit der zu Grunde liegenden Theorie dargestellt. Hauptergebnisse sind die ingenieurtechnische Entwicklung der Module aber auch eine Erweiterung des Punkt und Segmentbegriffs der klassischen euklidischen Geometrie. Diese Erweiterung wurde bei der Realisierung von planaren Nef-Polyedern benutzt und ihre Realisierbarkeit und Effizient damit nachgewiesen. Beide Softwaremodule sind in der entsprechenden Theorie verankert.

Der praktische Anteil dieser Arbeit basiert auf den zwei Softwarebibliotheken LEDA und CGAL und die entsprechenden Module sind auch darin integriert. Beide Softwarebibliotheken bieten Lösungen zu klassischen Problemen der algorithmische Geometrie, teilweise mit unterschiedlichen Philosophien. LEDA bietet einen einfacheren Zugang und folgt dabei einem monolithischen Entwurf der Datenstrukturen und Algorithmen. CGAL dagegen basiert auf aufwendigeren Techniken und realisiert die Idee generischer Programmierung. Der generische Ansatz basiert auf der intensiven Nutzung der C++-Template-Technologie. Weil diese sich nach wie vor weiterentwickelt und die entsprechenden Compiler noch lange nicht den definierten Technologiestandard erreichen, ist CGAL auch ein Softwareengineering-Experiment. Mit Hilfe der C++-Template-Technologie realisiert man schnittstellenbasierte generische Programmmodule, die zur Compilezeit zusammengesetzt werden können (Compilezeit-Polymorphismus). Dieser Ansatz vermeidet die entsprechenden Laufzeiteinbußen, die im klassischen objektorientierten Polymorphismus auftreten, wo Ableitungshierarchien und virtuelle Funktionen benutzt werden.

Der erste Teil dieser Arbeit beschreibt einen Geometriekern, der höherdimensionale euklidische Geometrie zur Verfügung stellt. Es werden die geometrischen Objekte, ihre Interaktion und die entsprechenden geometrischen Prädikate und Konstruktionen vorgestellt. Die sich daraus zusammensetzende Schnittstelle erlaubt die schnellere Realisierung von Algorithmen, denen höherdimensionale geometrische Problemstellungen zu Grunde liegen. Um eine flexible Benutzung zu ermöglichen, wurden Konzepte, wie z.B. Zahlentypen und intern verwendeten Algorithmen der lineare Algebra, herausgearbeitet und die generischen Schnittstellen und die ihnen entsprechenden Softwaremodelle dokumentiert. Letzteres erlaubt den Austausch der angebotenen Modelle, um in unterschiedlichen Anwendersituationen die optimale Realisierung der Konzepte zu ermöglichen. Um die Anwendbarkeit des Geometriekerns zu erhöhen und die Benutzung zu vereinfachen wurden die realisierten Kernfamilien (basierend auf Koordinatenrepräsentation) so entworfen, dass sie direkt als Traitsklassen in den entsprechenden Anwendungsprogrammen verwendet werden können. Dies ist ein allgemeines Softwareentwurfsmuster in CGAL ab Version 2.3.

Unser zweites Projekt befasst sich mit dem Entwurf und der Implementierung eines Datentyps zur Realisierung planarer Nef-Polyeder. Die zu Grunde liegende Theorie wurde von dem schweizer Mathematiker W. Nef verfasst. Unsere Realisierung benutzt planare Karten für die topologischen Beschreibung der Elemente eines planaren Nef-Polyeders und ihrer Inzidenzen. Die geometrische Komponente unseres Ansatzes liefert die geometrische Einbettung der Elemente und dient dazu, die auftretenden endlichen und unendlichen Strukturen in ihrer Behandlung zu vereinheitlichen. Dazu führen wir infimaximale Rahmen und damit zusammenhängend erweiterte Punkte und Segmente ein. Die Stärke unseres Ansatzes liegt einerseits in der klaren Trennung von topologischen und geometrischen Sachverhalten und andererseits in der möglichen transparenten Behandlung der erweiterten Objekte. Letztlich können wir zeigen, dass bekannte algorithmische Verfahren der affinen Geometrie auch auf unseren erweiterten geometrischen Objekten arbeiten. Aufbauend auf einem generischen Modul zur Überlagerung planarer Karten wird direkte Code-Wiederverwendung möglich. Die geometrisch unbegrenzte Ausdehnung von Nef-Polyedern wird durch unseren Ansatz symbolisch abgeschlossen und mit unserem generisches Standarverfahren können wir die binären Mengenoperationen des Datentyps, wie z.B. Schnitt und Vereinigung, relativ einfach realisieren.

Im zweiten Kapitel führen wir infimaximale Rahmen als einfache Erweiterung der affinen Geometrie ein. Die ausgearbeitete Theorie dient zur Realisierung der Nef-Polyeder, hat aber auch andere Anwendungen. Wir beschreiben drei verschiedene Implementierungen basierend auf Polynomarithmetik. Die Realisierungen unterscheiden sich hinsichtlich der Einfachheit der Umsetzung und ihrer Effizienz. Bei den ersten beiden (basierend auf homogener und kartesischer Komponentendarstellung) steht die programmiertechnische Überschaubarkeit im Mittelpunkt. Mit Hilfe eines Polynomzahlentyps, der zusammen mit den 2-dimensionalen CGAL Standardgeometriekernen benutzt wird, realisieren wir unsere erweiterten Objekte und die entsprechenden Prädikate. Die Korrektheit unseres Moduls basiert auf der Korrektheit der CGAL Kerne und unseres Zahlentyps. Letzterer realisiert algebraische Standardtechniken und ist daher einfach zu verifizieren. Die dritte Realisierung wiederum ist laufzeittechnisch optimiert. Die auftretenden algebraischen Ausdrücke werden explizit mit vielen Fallunterscheidungen ausprogrammiert und durch arithmetische Filtertechniken werden die Kosten von Langzahlarithmetikaufrufen möglichst minimiert.

Alle unsere Softwaremodule nutzen den Ansatz generischer Programmierung, d.h. man programmiert mit Konzepten und Modellen, wobei Modelle hier konkrete Realisierungen der abstrakten Konzepte sind. Leider existieren weder Sprachkonstrukte in C++, noch stellen die Compiler nennenswerte Hilfsmittel zur Verfügung, um die Übereinstimmung von Konzept und Modell zu testen. Daher erfordert generisches Programmieren einen entsprechenden Aufwand zur Dokumentation der verwendeten Schnittstellen. Diese Arbeit versucht den Entwurfsprozess komplett zu dokumentieren: die abstrakt erarbeiteten Konzepte, die entsprechenden Realisierungen (Modelle) derselben, und Auszüge aus der Dokumentation im Anhang. In den Implementierungsdarstellungen verwenden wir “literate programming” zur Dokumentation unserer Programme.

Was sind die Vorzüge des generischen Ansatzes? Im wesentlichen ermöglicht uns dieser die Erweiterung und Anpassung von Codemodulen, ohne diese zu kopieren und die Kopie dann zu verändern. Durch die inhärente Effizienz des Template-Mechanismus sind weiterhin Softwaretuning und Optimierung in einer klar definierten Art möglich. In Experimenten kann durch den Austausch entsprechender Modelle einfach auf optimierte Instanzen hingearbeitet werden.